
The 32-bit PA-RISC Run-time Architecture Document

HP-UX 11.0 Version 1.0

(c) Copyright 1997 HEWLETT-PACKARD COMPANY.

The information contained in this document is subject to change without 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 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.

**CSO/STG/STD/CLO
Hewlett-Packard Company
11000 Wolfe Road
Cupertino, California 95014**

By

The Run-time Architecture Team

CHAPTER 1	Introduction	7
	1.1 Target Audiences	7
	1.2 Overview of the PA-RISC Runtime Architecture Document	8
CHAPTER 2	Common Coding Conventions	9
	2.1 Memory Model	9
	2.1.1 Text Segment	9
	2.1.2 Initialized and Uninitialized Data Segments	9
	2.1.3 Shared Memory	10
	2.1.4 Subspaces	10
	2.2 Register Usage	10
	2.2.1 Data Pointer (GR 27)	10
	2.2.2 Linkage Table Register (GR 19)	10
	2.2.3 Stack Pointer (GR 30)	11
	2.2.4 Space Registers	11
	2.2.5 User-Readable Control Registers (CR 26 and CR 27)	11
	2.2.6 General Registers Summary	12
	2.3 External Naming Conventions	12
	2.4 Conventions for Accessing Data	13
	2.4.1 Static Variables	13
	2.4.2 C-Style Common	14
	2.4.3 Fortran-Style Common	14
	2.4.4 COBOL-Style Common	14
	2.4.5 Pascal Outer Block Globals	14
	2.4.6 Constants and Literals	14
	2.4.7 Automatic Variables	14
	2.4.8 Position-Independence	15
	2.5 Conventions for Calling Procedures	15
	2.5.1 Stack Frame Layout and Marker	15
	2.5.2 Stack frame after dynamic memory allocation	19
	2.5.3 Parameter Passing and Return Values	21
	2.5.4 Type Checking and Floating-Point Parameter Relocation	29
	2.5.5 Standard Procedure Calls	32
	2.5.6 Indirect Procedure Calls	36
	2.5.7 Millicode Procedure Calls	38
	2.6 Program Startup	39
CHAPTER 3	Relocatable Object Files	41
	3.1 Object File Header	43
	3.2 Compilation Unit Records	50
	3.3 Space Dictionary	52
	3.4 Subspace Dictionary	55
	3.5 String Areas	62
	3.6 Fixup Requests	62
	3.6.1 Fixup Rounding Modes	64
	3.6.2 Interpretation of rounding mode and field selector	65
	3.6.3 Examples of applying the rounding mode	67

-
- 3.6.4 Apply Fixups on instructions 68
 - 3.6.5 List of fixup requests 69
 - 3.6.6 Fixup opcodes, lengths and parameters 75
 - 3.6.7 Parameter Relocation Bits (rbits1, rbits2) 78
 - 3.7 Symbol Table 80

CHAPTER 4 Relocatable Libraries 97

- 4.1 Archive Header 98
- 4.2 Library Symbol Table Header Record 100
- 4.3 Library Symbol Table Format 104
 - 4.3.1 Symbol Directory 104
 - 4.3.2 SOM Directory 113
 - 4.3.3 Free Space List 114

CHAPTER 5 Executable Files 117

- 5.1 Object File Header 117
- 5.2 Auxiliary Headers 117
 - 5.2.1 Loader Auxiliary Headers 120
 - 5.2.2 Other Auxiliary Headers 121
- 5.3 Symbol Table 123
- 5.4 Stack Unwind Table 124
- 5.5 Recover Table 124
- 5.6 Auxiliary Unwind Table 124

CHAPTER 6 HP-UX Specifics 125

- 6.1 HP-UX Auxiliary Header 125
- 6.2 Program Startup 127
 - 6.2.1 Sample Assembly Listing of crt0 code 128
- 6.3 Shared Libraries 141
 - 6.3.1 Shared Library Memory Model 141
 - 6.3.2 Linkage Table 141
 - 6.3.3 The DL Header and Other Tables 143
 - 6.3.4 Version Auxiliary Header 148
 - 6.3.5 Import List 148
 - 6.3.6 Export Table 149
 - 6.3.7 Export Table Extension 152
 - 6.3.8 Shared Library List 153
 - 6.3.9 Module Table 155
 - 6.3.10 Shared Library Unwind Info 156
 - 6.3.11 String Table 158
 - 6.3.12 Dynamic Relocation Records 158
 - 6.3.13 Loading Shared Libraries 161
 - 6.3.14 Intra-library Version Control 163
 - 6.3.15 Library-Level Versioning 164
 - 6.3.16 Import and Export Stubs 167
- 6.4 System Calls 168

CHAPTER 7 Symbolic Debug Information 171

- 7.1 The Debug Information Organization 171
- 7.2 Compilation Unit Headers 172
 - 7.2.1 Basic typedef and structure definitions 172
 - 7.2.2 XDB Header structure definition: 174
- 7.3 Name and Type Tables 175
 - 7.3.1 File-class ("File") DNTT Entries 175
 - 7.3.2 Code-class ("Scoping") DNTT Entries 176
 - 7.3.3 Storage-class ("Name") DNTT Entries 183
 - 7.3.4 Type-class ("Type") DNTT Entries 187
 - 7.3.5 General ("overall") DNTT Entry Format 203
- 7.4 Static Analysis Information 205
 - 7.4.1 XREF Table (XT) Entry Format 205
 - 7.4.2 Static Analysis Support DNTT Entries 207
- 7.5 Source Line Table 208
 - 7.5.1 SLT Entry Format 208
 - 7.5.2 SLT Types and Data Structure 209
- 7.6 Value Table (VT) 211
- 7.7 Ordering of Table Entries 211
- 7.8 Postprocessing 212
- 7.9 Debug Format Changes for Debugging of Optimized Code (DOC) 216
 - 7.9.1 Debug Format Changes 216
 - 7.9.2 Object File Format Details 217
 - 7.9.3 Building the Line Tables 218
 - 7.9.4 Debug Format Changes 219
 - 7.9.5 Line Number Table Definition 222
 - 7.9.6 View/modify globals and arguments when safe 224

CHAPTER 8 Stack Unwind Library 227

- 8.1 Overview 227
- 8.2 Requirements for Stack Unwinding 228
 - 8.2.1 Unwinding Across an Interrupt Marker 229
 - 8.2.2 Unwinding from Stubs on HP-UX 229
 - 8.2.3 Unwinding from Millicode 229
 - 8.2.4 Instances in Which Unwinding May Fail 230
 - 8.2.5 Callee-Saves Register Spill 230
 - 8.2.6 Sample entry and exit code 230
- 8.3 Role of Stubs in Unwinding 232
 - 8.3.1 The Stub Unwind Types 233
 - 8.3.2 Unwinding from Parameter Relocation Stub 234
- 8.4 External Interface 236
 - 8.4.1 The Unwind Descriptor 236
 - 8.4.2 Unwind Utility Routines 240
 - 8.4.3 Initialize a Stack Unwind 247
 - 8.4.4 Unwind Examples: Using U_get_previous_frame 248
- 8.5 Setjmp and longjmp jmp_buf 256
- 8.6 Process Context 258

8.6.1	Ada Exception handling	258
8.6.2	C++ Exception handling	273

This document describes the runtime architecture for PA-RISC systems running either the HP-UX or the MPE/iX operating system. Other operating systems running on PA-RISC may also use this runtime architecture or a variant of it.

The runtime architecture defines all the conventions and formats necessary to compile, link, and execute a program on one of these operating systems. Its purpose is to ensure that object modules produced by many different compilers can be linked together into a single application, and to specify the interfaces between compilers and linker, and between linker and operating system.

The runtime architecture applies only to hardware platforms based on PA-RISC Revision 2.0.

The runtime architecture does not specify the application programming interface (API), the set of services provided by the operating system to the program. Thus, observing the runtime architecture does not automatically lead to a program that will run on all PA-RISC platforms. It does, however, allow many of the development tools to be shared to a large extent among the various operating systems.

When combined with a particular API, this runtime architecture leads to an application *binary* interface (ABI). In other words, an ABI can be regarded as the composition of an API, a hardware description, and a runtime architecture for that hardware.

1.1 Target Audiences

This document is intended for a variety of readers.

If you are a systems programmer, you will find information in this document describing the format of an executable object file, the memory model and startup environment

assumed by a valid program, and the architected interface between a program and the services provided by your operating system.

If you develop compilers or other development tools, you will find information in this document about calling conventions and other coding conventions, the object file format, interfaces to the linker, symbolic debug format, and other details important to program translation.

If you are an application programmer, this document can help you learn about the low-level details of how programs execute on PA-RISC. If you need to write assembly code, process object files, examine the stack, or perform dynamic linking, you will find the necessary information in this document.

1.2 Overview of the PA-RISC Runtime Architecture Document

Chapter 2 describes the coding conventions used by compilers and by assembly-language programmers. This includes details of the virtual memory model, usage of processor registers, external name conventions, addressing data, procedure calling and parameter passing, and the program startup environment.

Chapter 3 describes the format of relocatable object files, and Chapter 4 describes the format of relocatable libraries.

Chapter 5 describes the format of program files in general, while Chapters 6 cover details specific to the HP-UX operating systems, respectively. It also cover shared libraries and executable libraries.

Chapter 7 describes the format of the symbolic debug information generated by the HP compilers and used by the debugger.

Chapter 8 describes the details of stack unwinding, and the interfaces to the stack unwind library provided by HP.

Chapter 9 describes the library of millicode routines provided for the use of HP compilers.

Chapter 10 describes the principles of dynamic linking—that is, dynamically loading relocatable objects into the address space of a running process.

Common Coding Conventions

2.1 Memory Model

The PA-RISC virtual memory is a set of linear spaces. Each space is four gigabytes (2^{32} bytes) in size and is divided into four equal portions of one gigabyte (2^{30} bytes each), known as quadrants. The four quadrants in a space are numbered 0,1,2, and 3, from low memory to high memory. An application can address 2^{16} spaces. Each application has its own short address space composed of these four distinct quadrants (can possibly be four distinct spaces).

2.1.1 Text Segment

The first quadrant (quadrant 0) of the short address space is mapped by space register 4 to the first quadrant of a space containing the shared text. The text is readable and executable, but not writable and must begin at a page boundary. An application must not change the contents of space register 4.

This area of memory is used to store code (machine instructions), and literals only. The text address begins at 0x00000000 and ends at 0x3FFFFFFF.

2.1.2 Initialized and Uninitialized Data Segments

The second quadrant (quadrant 1) of the short address space is mapped by space register 5 to the second quadrant of a space containing the private data of applications. The data section is readable, writable, and executable and must begin at a page boundary. The private data includes the initialized data, the uninitialized data (BSS), the heap and the user stack.

Data segments start at 0x40000000 and end at 0x7FFFFFFF.

2.1.3 Shared Memory

The third and fourth quadrant (quadrant 2 and 3) of the short address space is mapped by space register 6 and 7 to quadrants containing shared memory. Those portions of the shared memory that have been legally attached to the process via shared data memory system calls are readable and writable. The upper 256 megabytes of the fourth quadrant is not readable, writable, or executable by applications. The first page of the fourth quadrant is the Gateway page.

Shared memory starts at 0x80000000 and ends at 0xFFFFFFFF.

2.1.4 Subspaces

While a space is a fundamental concept of the architecture, a subspace is just a logical subdivision of a space. The linker groups subspaces into spaces as it builds an executables program file. On HP-UX systems, all subspaces in the code space must be in quadrant 0, and all subspaces in the data space must be in quadrant 1.

2.2 Register Usage

2.2.1 Data Pointer (GR 27)

By software convention, general register GR 27 is used to point to the beginning address of global data in the data segment (\$PRIVATE space).

The start up code for each process sets up this address which is also known as the address of symbol \$global\$. Compilers and the linker then use this symbol to assign global data or to relocate data addresses.

2.2.2 Linkage Table Register (GR 19)

The general purpose caller-saves register GR 19 has a special meaning in HP-UX shared library. In an HP-UX shared library, register GR 19 is used for the **Data Linkage Table**.

Each shared library and incomplete executable contains a linkage table, which is allocated in the DATA space for the file. The linkage table is divided into two parts, the **Data Linkage Table** (DLT), and the **Procedure Linkage Table** (PLT). The PLT contains an entry for each unresolved procedure symbol referenced within the object and it is placed immediately following the DLT (if one exists).

The DLT contains an entry for each data or procedure symbol that is accessed indirectly. Each DLT entry is a single word which contains a pointer to the actual data item referenced indirectly; this pointer value is assigned by the dynamic loader, after mapping the shared library. All references to data items go directly through the DLT and GR 19 is

reserved to point to the middle of this table. The linker allocates GR 19-relative offsets for each DLT entry, and uses those offsets when applying fixups.

2.2.3 Stack Pointer (GR 30)

Because no explicit procedure call stack exists in the PA-RISC processor architecture, the stack is defined and manipulated entirely by software convention. By convention, GR 30 is used for the stack pointer.

The stack pointer always points to the first unused byte of data segment beyond the stack frame marker, and is 64-byte aligned.

When a process is initiated by the operating system, a virtual address range is allocated for that process to be used for the call stack, and the stack pointer (GR 30) 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 frame to exist at the address below the stack pointer. When procedures are exited, the stack pointer is decremented by the same amount.

2.2.4 Space Registers

The following table (table 1) summarizes the PA-RISC available space registers and their usage.

TABLE 1

Space Register Usage

Register Name	Other Names	Usage Convention
SR 0		Caller-saves space register or millicode return space register.
SR 1	sarg sret	Space argument and return register or caller-saves space register.
SR 2		Caller-saves space register.
SR 3		Callee-saves space register.
SR 4		Code space register (stubs save and restore on inter-module calls).
SR 5		Data space register, modified only by privileged code.
SR 6		System space register, modified only by privileged code.
SR 7		System space register, modified only by privileged code.

2.2.5 User-Readable Control Registers (CR 26 and CR 27)

CR27 is used to point to the beginning address of thread specific data, CR26 is not used at this point.

2.2.6 General Registers Summary

The following table (table 2) summarizes general register usage:

TABLE 2 **General Register Usage**

Register Name	Other Names	Usage Convention
GR 0		Zero value register. (Writing to this register does not affect its contents.)
GR 1		Scratch register (caller-saves). (can be destroyed by call mechanism).
GR 2	RP	Return pointer and scratch register.
GR 3 - GR 18		General purpose callee-saves registers.
GR 19		Shared Library linkage register.
GR 19 - GR 22		General purpose caller-saves registers.
GR 23	arg3	Argument register 3 or general purpose caller-saves register.
GR 24	arg2	Argument register 2 or general purpose caller-saves register.
GR 25	arg1	Argument register 1 or general purpose caller-saves register.
GR 26	arg0	Argument register 0 or general purpose caller-saves register.
GR 27	DP	Global data pointer; may not be used to hold other values. (Stubs save and restore on inter-module calls)
GR 28	ret0	Function return register on exit or function result address on entry. May also be used as a general purpose caller-saves register.
GR 29	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.
GR 30	SP	Stack pointer, may not be used to hold other values.
GR 31		Millicode return pointer, Scratch register (caller-saves).

2.3 External Naming Conventions

The external naming conventions (commonly known as *name space pollution solution*, or *secondary definitions*) are designed to allow ANSI C, POSIX users to define their own versions of reserved symbols, while still allowing users to access the underlying system symbols if they want to.

The external naming conventions provide a secondary definition for special names (code or data) that would be specified from within the library source code by means of a pragma, for example:

```
#pragma _HP_SECONDARY_DEF _open open

{
    /* code for open */
}
```

open = secondary symbol
_open = primary symbol

Since open is only a secondary definition within libc, a primary definition of open provided by the user can override it. Within libc itself, _open is called directly to avoid conflicts with the user version's of open.

In implementing the secondary definitions, the linker makes the following assumptions:

- Secondary definitions would be used only by internal developers of libc and libm.
- The reference to a secondary definition must be seen before any definition of that symbol.
- No modules within libc or libm will make references to secondary definitions.
- Secondary symbol definitions will be ignored if there are no outstanding references to them. Secondary symbols that are not used to resolve references will not be placed to the output file, and secondary symbols that are used to resolve references will have the *secondary_def* flag cleared in the resultant output file.

2.4 Conventions for Accessing Data

This section describes the various classes of data, how they are mapped into the memory model, and how the program should address that data.

2.4.1 Static Variables

Static variables can be initialized or uninitialized, they can be small or large, and they can be local or global scope. In general, compilers allocate global variables relative to Data Pointer (DP) or GR 27, and allocate local variables relative to Stack Pointer (SP) or GR 30. Static variables are allocated in the DATA subspace of PRIVATE space. Please refer “Symbol Table” on page 80 for details of symbol scopes and symbol types.

Local variables are managed by the compilers and are not visible in the object files.

Initialized global data are defined by symbols whose scope are *universal*.

The following code segments are used to make references to common, *NOT position independent* globals:

- To form the address of global X into register RR:

```
ADDIL    LR'X-$global$, DP
LDO      RR'X-$global$ (r1), RR
```

- To load the global X into register RR

```
ADDIL    LR'X-$global$, DP
LDW      RR'X-$global$ (0, r1), RR
```

LR' and RR' are representing fixups of type R_DP_RELATIVE emitted for global X in the above code segments.

Three aspects must be described: (1) the coding conventions to be followed by the compiler or in assembly code, (2) allocating the data to the correct segment, and (3) the responsibilities of the linker in relocating or transforming the code and allocating the data

2.4.2 C-Style Common

Uninitialized external-scope variables in C, without the *extern* keyword, are normally implemented similarly to Fortran Common blocks. The variables are treated as imported symbols, but are allocated automatically by the linker if no definition for the symbol is found in the program

2.4.3 Fortran-Style Common

Fortran Common blocks differ from C-style common only because the linker needs the ability to extend an initialized common block if an uninitialized declaration for the common block is larger than the initialized definition.

2.4.4 COBOL-Style Common

2.4.5 Pascal Outer Block Globals

Pascal has two methods for allocating global variables. In one method, the compiler allocates the global variables and assigns fixed dp-relative addresses to each symbol. Since the compiler sees the entire set of outer block declarations in each separate compilation, no link-time allocation is necessary, and the global variable names do not need to be externally visible.

In the second method, Pascal global variables are treated as in C.

2.4.6 Constants and Literals

Constant data and compiler-generated literals can be allocated in the text segment, or they can be allocated as static variables. Data allocated in the text segment must be accessed in a different fashion than data in the data segment, so there must be some support for determining which form of code generation to use when making an external reference to data whose allocation is unknown at compile time. Currently, the compiler assumes that constant data items are declared consistently at definition and reference sites.

2.4.7 Automatic Variables

Most local variables are allocated in the procedure stack frame, and are assigned fixed sp-relative offsets at compile time. These variables are not visible in the object files and no link time relocation (fixups) are needed.

Example assembly code uses to access local integer X:

```
LDW      -offset(0, R30), tmp1; to load X into register tmp1.
```

offset is assigned by the compiler.
STW tmp1, -offset(0, R30); to store X back to memory.

2.4.8 Position-Independence

In *position independent* compilation, the data linkage register (GR 19) and T' fixup will be used to access global variables. Depending on the size of the DLT table, short or long form code sequences will be generated.

- If the size of the DLT table is less than or equal to 16K bytes, the following code sequence will be used to form the address of a variable or to load the content of a variable, respectively:

LDW T' X(0,R19), tmp1
LDO offset(tmp1), RR; Omit if offset = 0, RR is used instead of tmp1.

and

LDW T' X(0,R19), tmp1
LDW offset(0,tmp1), RR

Note that the 16K bytes restriction on the DLT size are imposed because the T' fixup on the LDW allows for a 14-bit signed offset only.

The T' fixup specifier should generate a DLT_REL fixup proceeded by an FSEL override fixup.

- If the DLT table size is greater than the 16K bytes limit, the linker will emit an error indicating to users that this program must be recompiled with the +Z option. The +Z option produces the following long form code sequence:

To form the address of a variable:

ADDIL LT' X, R19
LDW RT' X(0, R1), tmp1
LDO offset (tmp1), RR; Omit if offset = 0 and RR is used instead of tmp1.

To load the content of a variable:

ADDIL LT' X, R19
LDW RT' X(0, R1), tmp1
LDW offset (0, tmp1), RR;

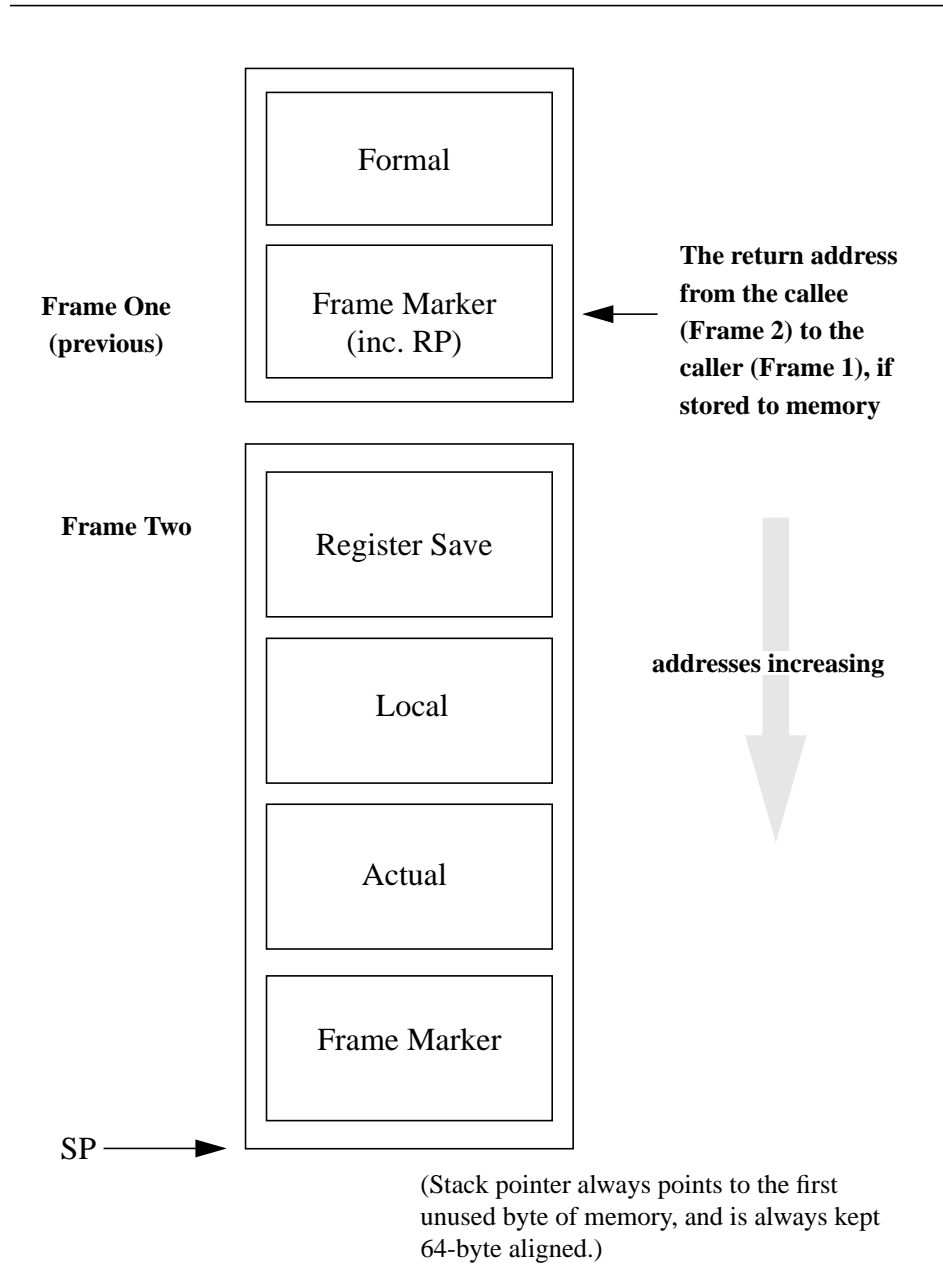
2.5 Conventions for Calling Procedures

2.5.1 Stack Frame Layout and Marker

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 the 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. 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 below:



The elements of a single stack frame that must be present in order for a procedure call to occur are shown below in Table 3. 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.

TABLE 3

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 be 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.

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.

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.

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 be unused.

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 table 3, 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.

2.5.2 Stack frame after dynamic memory allocation

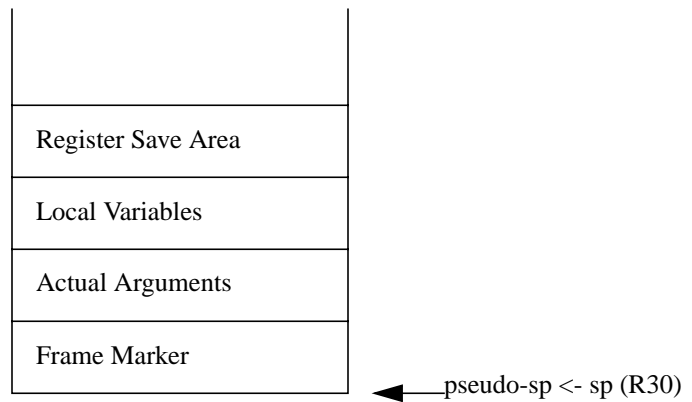
This section describes the extension of the PA-RISC Procedure Calling Convention to allow C routines to allocate memory on the stack using the built-in `alloca()` routine. `Alloca()` is a routine that works like `malloc()` except that it allocates storage from the

stack instead of the heap. The storage will be freed automatically when the routine that called `alloca()` exits or returns. The following is the declaration of the `alloca()` routine:

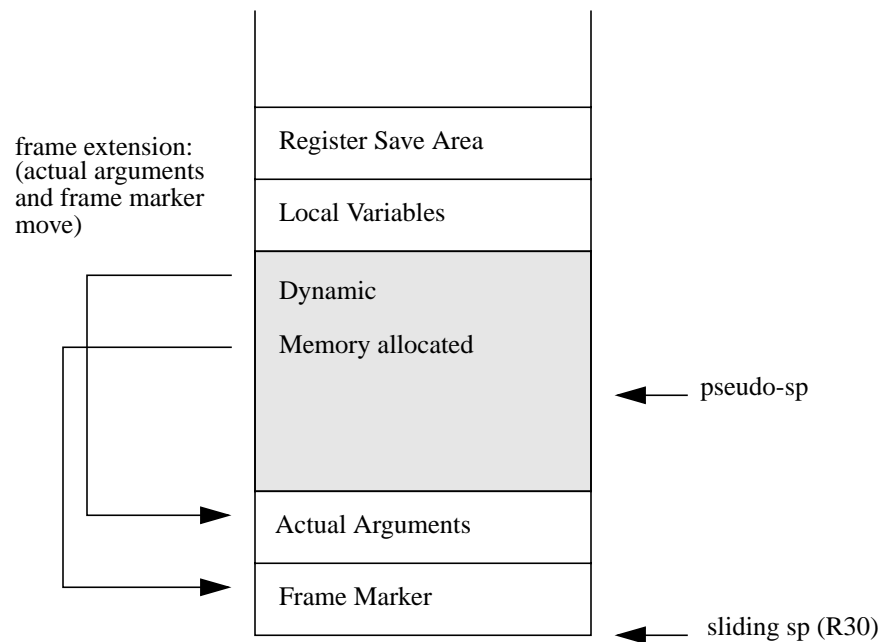
```
char *alloca(int Size)
```

Here is what the stack looks like as it allocates dynamic memory:

Initially:



After first `alloca()`:



Since the stack pointer, SP, is modified for each call to `alloca()`, the existing unwind mechanism needs to be enhanced. Refer to the unwind chapter for details of how the entry and exit code are changed to support the variable frame size. Also, refer to that chapter for details of how `alloca()` works on PA-RISC.

2.5.3 Parameter Passing and Return Values

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.

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.

GR0	Value (zero)
GR1	Scratch *
GR2	RP (Return Pointer/Address)
GR3	Callee Saves
:	
:	
GR18	
GR19	Caller Saves
:	
GR22	
GR23	Arguments *
:	
GR26	
GR27	DP (Global Data Pointer)
GR28	Return Values *
GR29	
GR30	SP (Stack Pointer)
GR31	MRP (Millicode Ret. Ptr)/Scratch *

*** May also be considered part of the caller-saves partition**

Figure 2-2: Register Partitioning

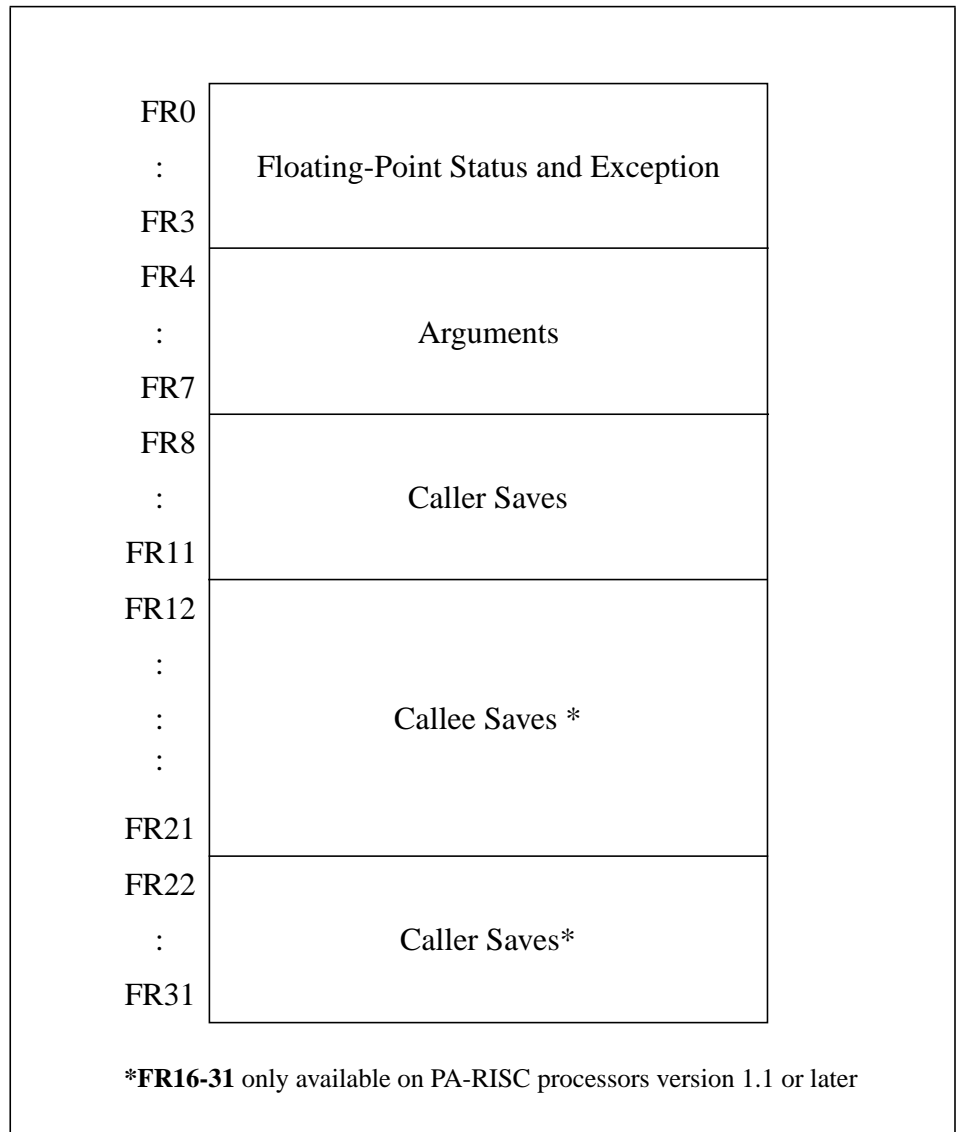


Figure 2-3: Floating-Point Registers

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 Processor Status Word (PSW).
- The state, including internal registers, of any special function units accessed by the architected SPOP operations.

The following is not guaranteed to be preserved across calls:

- The shift (cr11) or any control registers that are modified by privileged software (e.g. Protection IDs).

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.

Note



If the routine in question is a non-leaf routine, return pointer 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.

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 4 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.

TABLE 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

Inter-Language Parameter Data Types and Sizes

- 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.

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.

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.

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.

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 requirement simply 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 5.

TABLE 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)

Function Return Values

Function result values are placed in registers as described in Table 6. 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 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) ^a
Floating-Pt. (64-bit)	fret (FR4) ¹
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 ^b

a. 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.)

b. The caller may not assume that the result's address is still in GR28 on return from the function.

2.5.4 Type Checking and Floating-Point Parameter Relocation

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.

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 by the linker. 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.

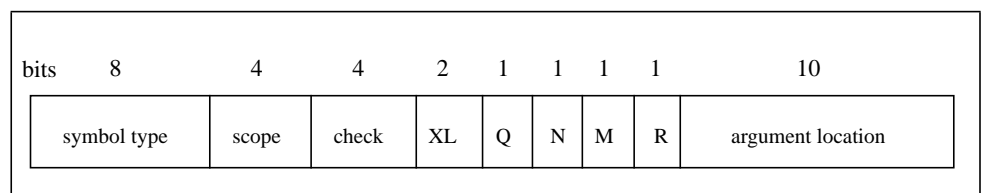


Figure 2-4: Layout of Symbol Definition Record

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) ^a
11 frupper	Floating-point register (bits 32..63) ¹

a. 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 2-5.

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".)

The linker 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

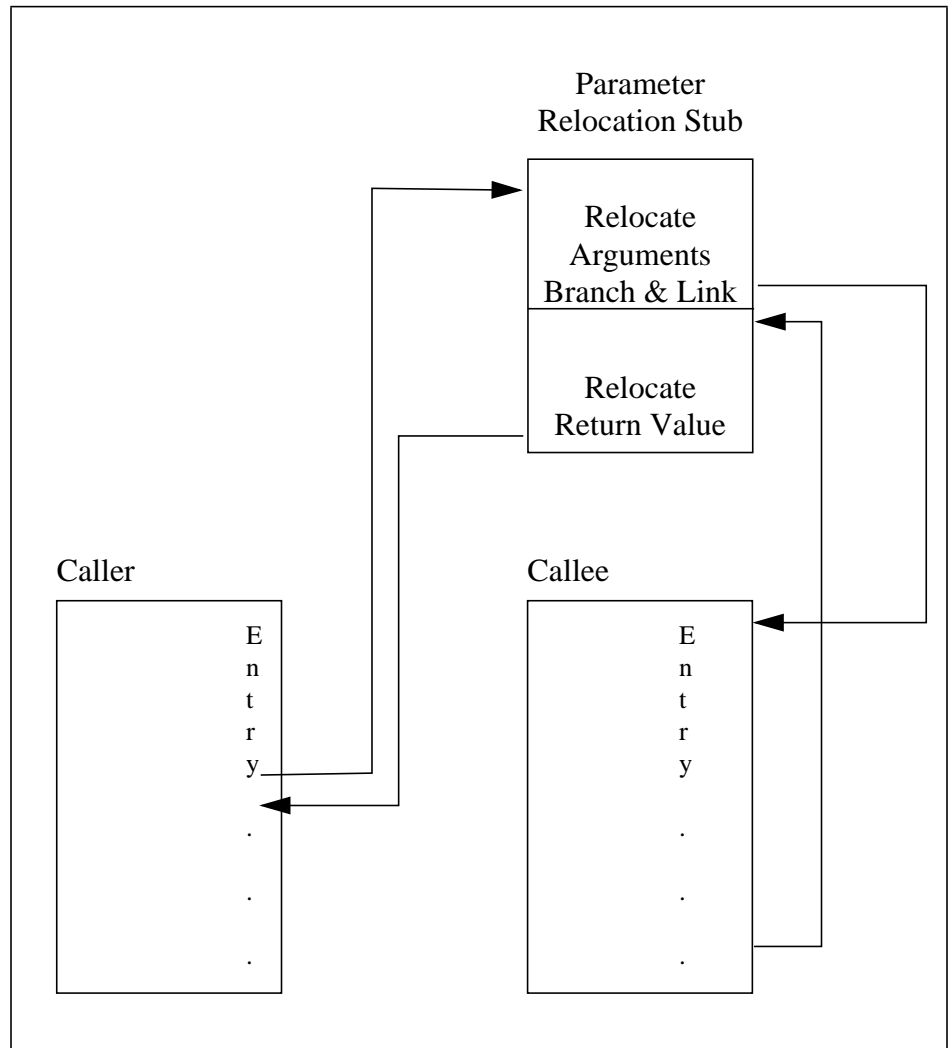


Figure 2-5: Parameter Relocation Stub.

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.

2.5.5 Standard Procedure Calls

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.

Long Calls

Normally, the compilers generate a single-instruction call sequence using the BL instruction to perform a procedure call. However, the compilers can be forced to generate a long call sequence when the module is so large that the BL instruction is not guaranteed to reach the beginning of the subspace. For example, COBOL compilers that typically compile large applications need to make sure that the BL instruction can reach to the beginning of subspace (including an estimation of stubs that will be generated by the linker, currently HP compilers allocate 2K bytes for stubs) is within the range of 256K bytes. Otherwise, a long call sequence as show below should be generated instead of the BL branch instruction. At link phase, the linker can then insert a stub. The existing long call sequence is three instructions, using an absolute target address:

```
LDIL      L'target,%r1
BLE       R'target(%sr4,%r1)
COPY      %r31,%rp
```

When the PIC option is in effect, the compilers must generate the following instruction sequence, which is PC-relative:

```
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
```

External Calls

External calls occur 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. 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 must generate 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.

Code which is used in a shared library must be compiled as position independent code. Refer to compiler documentation for specific instructions. Code in the program file is not PIC and the linker places the import/export stubs into the program file to handle external calls.

When building a shared library, the linker must generate import and export stubs for all procedures which can be called from outside of the shared library. Figure 2-6 below shows the control flow of an external call.

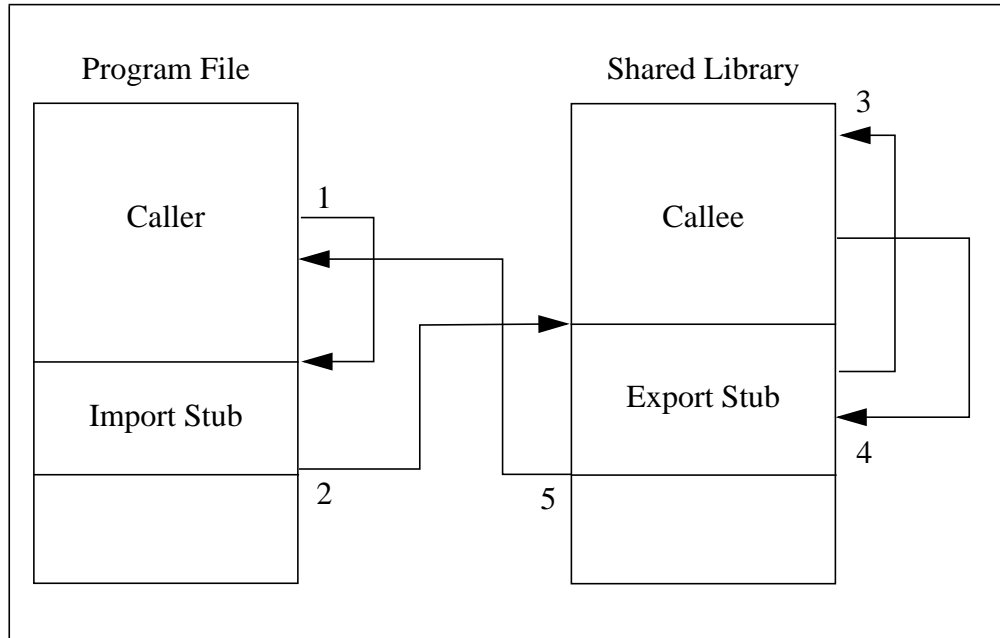


Figure 2-6: Flow of an External Procedure Call

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 external call performs the following steps:

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

```
LDW disp(0, dp), r21
```

1. The two import stubs are for +DA2.0. For +DA1.1, the compiler uses LDSID/MTSP/BE sequence instead of BVE.

```
LDW disp+4(0, dp), r19
BVE 0(r21)
STW rp, -24(0, sp)
```

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 (calling) stub used in a shared library is shown below:

Import Stub (Shared Library)

```
X':   LDW disp(0, r19), r21
      LDW disp+4(0, r19), r19
      BVE 0(r21)
      STW rp, -24(0, sp)
```

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:

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

The code sequence of the export stub is shown below:

For +DA1.1 :

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 register
	BE,N	0(sr0,rp)	; inter-space return

For +DA2.0 :

```

X':  <optional parameter relocation code>
      BLL      <entry>
      NOP
      <optional return relocation code>
      LDW      -24(0,sp),rp          ; restore the original RP
      BVE,N    0(rp)                ; inter-space return
  
```

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.

2.5.6 Indirect Procedure Calls

Procedure Labels and Dynamic Calls

PA-RISC compilers must 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 in Figure 2-7.

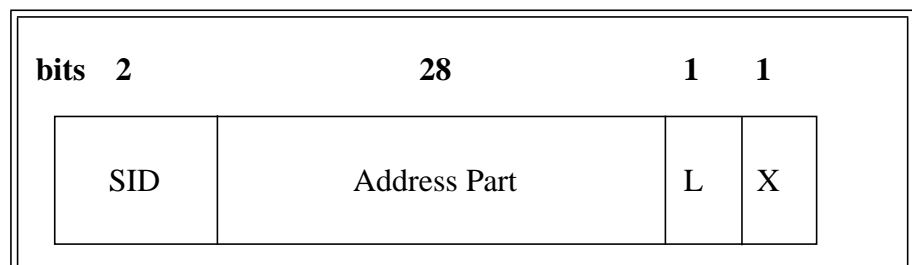


Figure 2-7: Procedure Label Layout

The X field in the address section of the procedure label is reserved. 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 (linked in automatically by linker) 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.

2.5.7 Millicode Procedure Calls

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).

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.

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:

- The return address (MRP) is passed in gr31; and
- 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:

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

2.6 Program Startup

All programs must include the start-up routine `crt0.o`. This code defines entry points, initializes program variables such as `DP`, and checks for dynamic libraries. The symbols defined by `crt0.o` are listed in Table 7, and the value of processor's registers are defined in Table 8.

TABLE 7 Symbols Defined By `crt0.o`

Symbol	Description
<code>__argc_value</code>	A variable of type <code>int</code> containing the number of arguments.
<code>__argv_value</code>	An array of character pointers to the arguments themselves.
<code>_environ</code>	An array of character pointers to the environment in which the program will run. This array is terminated by a null pointer.
<code>_SYSTEM_ID</code>	A variable of type <code>int</code> containing the system id value for an executable program.
<code>\$START\$</code>	Execution start address.
<code>_start</code>	A secondary start-up routine for C programs, called from <code>\$START\$</code> , which in turn calls <code>main</code> . This routine is contained in the C library rather than in the <code>crt0.o</code> file. For Pascal and FORTRAN programs, this symbol labels the beginning of the outer block (main program) and is generated by the compilers.
<code>\$global\$</code>	The initial address of the program's data pointer. The start-up code loads this address into GR 27.
<code>\$UNWIND_START</code>	The beginning of the stack unwind table.
<code>\$UNWIND_END</code>	The end of the stack unwind table.
<code>\$RECOVER_START</code>	The beginning of the try/recover table.
<code>\$RECOVER_END</code>	The end of the try/recover table.
<code>__text_start</code>	The beginning address of the program's text area. ^a
<code>__data_start</code>	The beginning address of the program's data area. ^a

a. The symbols `__text_start` and `__data_start` are defined by the linker.

TABLE 8 **Register Definition at Process Initialization**

Register	C Source Definition	Value
GR 24	char ** envp	array of pointers to environment strings
GR 25	char ** argv	array of pointers to arguments
GR 26	int argc	argument count
GR 30		stack pointer, set by O.S.
All Other GR's		Undefined
SR 4		address of first quadrant of virtual address space ^a
SR 5		address of second quadrant of virtual address space ^b
SR 7		address of fourth quadrant of virtual address space
SR0-SR3		Undefined
SAR (Shift Amount Register)		Undefined
All Co-processors' Registers		Undefined
CCR (Co-processor Config. Register)		If any bits are set then the corresponding co-processor must be present and functional
PSW (Processor Status Word)		Bits ^c C,D,P,Q =1; Bits B, M, N =0

a. Space register 4 is unprivileged, but it must not be modified by a conforming application.

b. Space registers 5 and 7 are privileged and cannot be modified by a conforming application.

C=Code Addr. Translation Enable,
D=Data Addr. Translation Enable
P=Protection ID Validation Enable
Q=Interrupt State Collection Enable
B=Taken Branch bit
M=High-priority machine check mask
N=Nullify bit

c.

The SOM common object file format defined in this document is intended to be a common representation of code and data for all compilers which generate code for PA-RISC based systems. A SOM is the smallest unit which may be generated by a compiler, and it may exist as a single entity or as part of a collection.

The SOM consists of a main header record, an exec auxiliary header record, and other optional components. The location and size of the auxiliary header record and all other components are defined in the main header record. Each location is given by a byte offset (relative to the first byte of the header), and the size is given either by the number of entries (records) of the component, or the total number of bytes in the component.

The first byte of the header record is also the first byte of the SOM. It contains a 'magic' number which distinguishes the SOM from any other entity, such as a Library File or a random access archive. In addition to defining the size and location of the other components of the SOM, the header contains a time stamp and other identifying information.

Figure 3-1 below shows the general block diagram of a SOM.

Table 9 shows a suggested layout of records in a SOM.

Figure 2-8: Block Diagram of the SOM

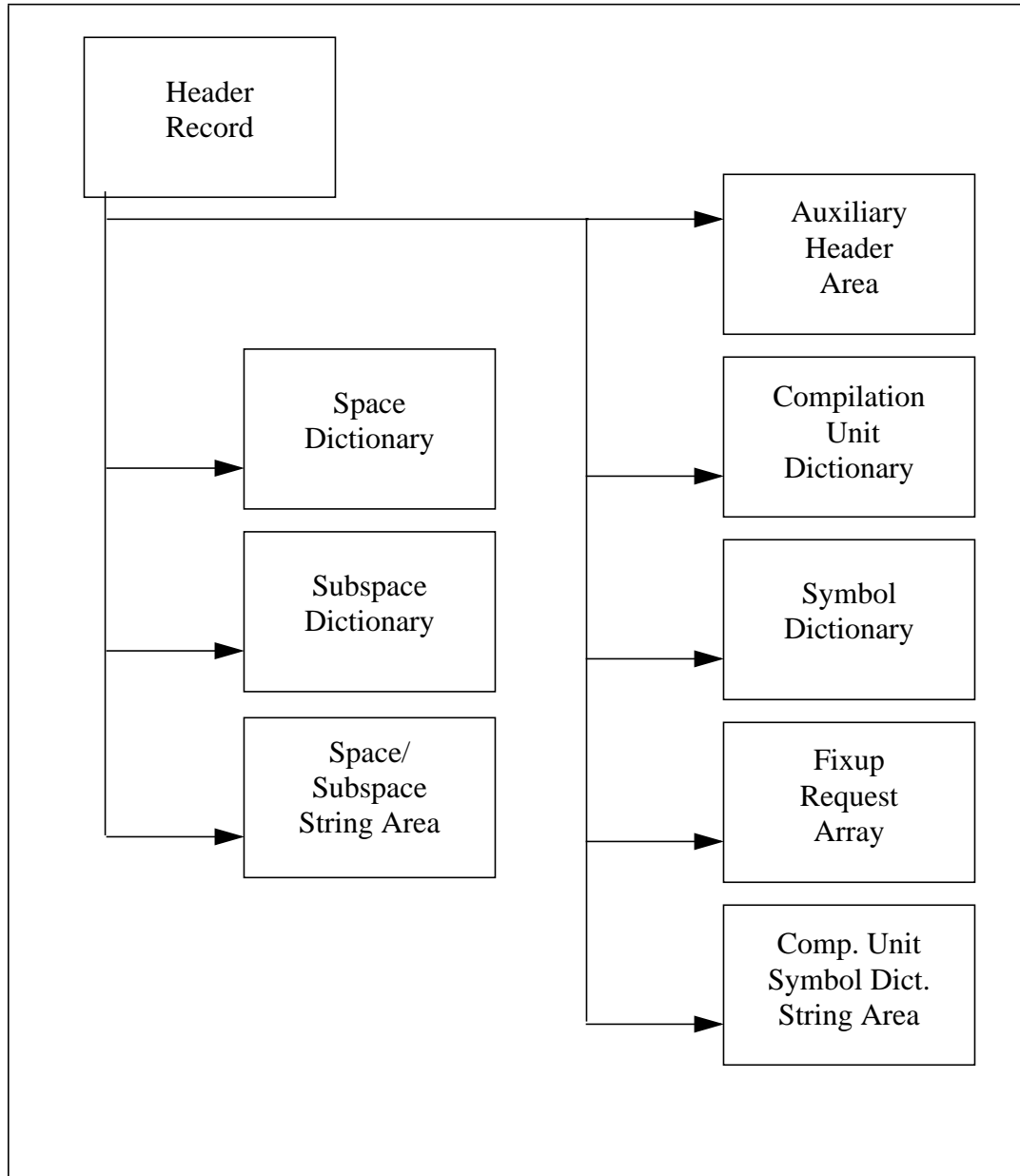


TABLE 9 Record Layout of a SOM

Header Record
Auxiliary Header Record
Space Records
Subspace Records
Loader Fixup Records
Space Strings
Symbol Records
Fixup Records
Symbol Strings
Compiler Records
Data for Loadable Spaces
Data for Unloadable Spaces

3.1 Object File Header

The first halfword of the header record contains a 'system id' number, identifying the target architecture of the SOM. The second halfword of the header record contains a 'magic number', identifying the type of this SOM. Following this, a character array will contain the version ID of the SOM format, and a time stamp specifying the time of creation of the particular SOM.

The remaining fields in the header record define the other components of the SOM. These fields provide a means to do bounds checking when there is a reference to a particular component.

The SOM header is required in any executable or relocatable object.

The C language definition of the SOM header is shown in Figure 2-9

system_id

This 2-byte field is used to identify the architecture that this object module is targeted for. The system ID for PA-RISC 1.0 systems is 20b (hexadecimal) , for PA-RISC 1.1 systems is 210 (hexadecimal), and for PA-RISC 2.0 systems is 214 (hexadecimal).

a_magic

This 2-byte field is a number that indicates certain characteristics about the internal format of the object module. The magic numbers that are currently defined for use on PA-RISC systems are listed in Table 10.

Figure 2-9: Definition of SOM Header Fields

```

struct header {
    short int    system_id;        /* magic number - system */
    short int    a_magic;          /* magic number - file type */
    unsigned int version_id;        /* version id; format=YYMMDDHH */
    struct       sys_clock file_time; /* system clock- zero if unused */
    unsigned int entry_space;      /* index of space containing
                                   entry point */
    unsigned int entry_subspace;   /* index of subspace for
                                   entry point */
    unsigned int entry_offset;     /* offset of entry point */
    unsigned int aux_header_location; /* auxiliary header location */
    unsigned int aux_header_size;  /* auxiliary header size */
    unsigned int som_length;       /* length in bytes of entire som */
    unsigned int presumed_dp;      /* DP value assumed during
                                   compilation */
    unsigned int space_location;   /* location in file of space
                                   dictionary */
    unsigned int space_total;      /* number of space entries */
    unsigned int subspace_location; /* location of subspace entries */
    unsigned int subspace_total;   /* number of subspace entries */
    unsigned int loader_fixup_location; /* MPE/iX loader fixup */
    unsigned int loader_fixup_total; /* number of loader fixup records */
    unsigned int space_strings_location; /* file location of string area
                                   for space and subspace names */
    unsigned int space_strings_size; /* size of string area for space
                                   and subspace names */
    unsigned int init_array_location; /* reserved for use by system */
    unsigned int init_array_total;    /* reserved for use by system */
    unsigned int compiler_location;   /* location in file of module
                                   dictionary */
    unsigned int compiler_total;      /* number of modules */
    unsigned int symbol_location;     /* location in file of symbol
                                   dictionary */
    unsigned int symbol_total;        /* number of symbol records */
    unsigned int fixup_request_location; /* location in file of fix_up
                                   requests */
    unsigned int fixup_request_total; /* number of fixup requests */
    unsigned int symbol_strings_location; /* file location of string area
                                   for module and symbol names */

```

Figure 2-9: Definition of SOM Header Fields (Continued)

```

unsigned int    symbol_strings_size;    /* size of string area for
                                         module and symbol names    */
unsigned int    unloadable_sp_location; /* byte offset of first byte of
                                         data for unloadable spaces */
unsigned int    unloadable_sp_size;    /* byte length of data for
                                         unloadable spaces    */
unsigned int    checksum;
};

```

Note that the magic numbers for executable and relocatable SOM libraries indicate that the header is an LST header rather than a SOM header.

TABLE 10**Magic Number Values**

Magic Number (in hexadecimal)	SOM Type
0104	Executable SOM Library
0106	Relocatable SOM
0107	Non-sharable, executable SOM
0108	Sharable, executable SOM
010B	Sharable, demand-loadable executable SOM
010D	Dynamic Load Library
010E	Shared Library
0619	Relocatable SOM Library

version_id

This is a number that is used to associate the SOM with the correct definition of its internal organization. The value of the number will be an encoding of the date the SOM version was defined.

The version ID can be interpreted by viewing it in its decimal form and separating it into character packets of YYMMDDHH, where YY is the year, MM is the month, DD is the day, and HH is the hour.

The version_id that are currently defined for use by conforming applications are 85082112 for old version ID and 87102412 for new version ID with new fixups.

file_time

The file time is a 64 bit value that represents the time the file was last modified. The file time is actually composed of two 32 bit quantities where the first 32 bits is the number of seconds that have elapsed since January 1, 1970 (at 0:00 GMT), and the second 32 bits is the nano second of the second (which requires 30 bits to represent).

This value is independent of any modification time maintained by other subsystems (e.g. the file system). The use of this field is optional, but if it is not used it will be set to zero.

entry_space

This is the space dictionary index of the space containing the main entry point of this particular SOM.

entry_subspace

This is the subspace dictionary index of the subspace containing the main entry point of this particular SOM.

entry_offset

This is the byte offset of the main entry point of the SOM relative to the first byte of the space.

aux_header_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the auxiliary header area. Setting all bits to zero indicates that the auxiliary header record is not defined in a SOM. The auxiliary header must start on a word boundary. *Aux_header_location* must have a value in the range 0 to $2^{31}-1$. See “Auxiliary Headers” on page 117. for restrictions on auxiliary headers.

aux_header_size

This field contains the byte length of the auxiliary header area. If the number of bytes is zero it indicates that no auxiliary headers are defined in the SOM. The size must be a multiple of 4 bytes. The field *aux_header_size* must have a value in the range 0 to $2^{31}-1$.

som_length

This field contains the length in bytes of the entire SOM. The field *som_length* must be in the range 0 to $2^{31}-1$.

presumed_dp

This field is only specified for shared libraries. It contains the value of the data pointer (DP) assumed during compilation or linking of this SOM. In a shared library, *presumed_dp* is the value of the data pointer that the linker used as a base to initialize data. The dynamic loader will subtract this value to get the offset of the data.

space_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the space dictionary. Setting all bits to zero in *space_location* indicates that the space dictionary is not defined in a SOM. The space dictionary must start on a word boundary. *Space_location* must have a value in the range 0 to $2^{31}-1$.

space_total

This field contains the number of space records in the space dictionary. Setting all bits to zero in *space_total* means that the space dictionary is not defined in a SOM. *Space_total* must have value in the range 0 to $2^{31}-1$.

subspace_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the subspace dictionary. Setting all bits to zero in *subspace_location* indicates that the subspace dictionary is not defined in a SOM. The subspace dictionary must start on a word boundary. *Subspace_location* must have a value in the range 0 to $2^{31}-1$.

subspace_total

This field contains the number of subspace records in the subspace dictionary. Setting all the bits to zero in *subspace_total* means that the subspace dictionary is not defined in a SOM. *Subspace_total* must have a value in the range 0 to $2^{31}-1$.

loader_fixup_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the loader fixup array. Loader fixup is used only in MPE/iX for relocation at load time. Setting all bits to zero in *loader_fixup_location* indicates that the loader fixup array is not defined in the SOM. The loader fixup array must start on a word boundary and the *loader_fixup_location* must have a value in the range 0 to $2^{31}-1$.

loader_fixup_total

This field contains the number of loader fixup records in the loader fixup array. Setting all bits to zero in *loader_fixup_total* indicates that the loader fixup array is not defined in the SOM. *loader_fixup_total* must have a value in the range 0 to $2^{31}-1$.

space_strings_location

Space_strings_location points to a string area that contains both space and sub-space names. It is a byte offset relative to the first byte of the SOM header. Setting all bits to zero indicates that the space subspace string area is not defined in a SOM. The space subspace string area must start on a word boundary. *Space_strings_location* must have a value in the range 0 to $2^{31}-1$.

space_strings_size

This field contains the byte length of the space subspace string area. Setting all bits to zero in *space_strings_size* indicates that the string area is not defined in a SOM. *Space_strings_size* must be a multiple of 4 bytes and be in the range 0 to $2^{31}-1$.

init_array_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the initialization pointer array. Setting all bits to zero in *init_array_location* indicates that the initialization pointer array is not defined in the SOM. The initialization pointer array must start on a word boundary and the *init_array_location* must have a value in the range 0 to $2^{31}-1$.

init_array_total

This field contains the number of initialization pointer records in the initialization pointer array. Setting all bits to zero in *init_array_total* indicates that the initialization pointer array is not defined in the SOM. *init_array_total* must have a value in the range 0 to $2^{31}-1$.

compiler_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the compilation unit dictionary. Setting all bits to zero in *compiler_location* indicates that the compilation unit dictionary is not defined in a SOM. The compilation unit dictionary must start on a word boundary. *Compiler_location* must have a value in the range 0 to $2^{31}-1$.

compiler_total

This field contains the number of compilation unit records in the compilation unit dictionary. Setting all bits to zero in *compiler_total* means that the compilation unit dictionary is not defined in a SOM. *Compiler_total* must have a value in the range 0 to $2^{31}-1$.

symbol_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the symbol dictionary. Setting all bits to zero in *symbol_location* indicates that the symbol dictionary is not defined in a SOM. The symbol dictionary must start on a word boundary. *Symbol_location* must have a value in the range 0 to $2^{31}-1$.

symbol_total

This field contains the number of symbol records in the symbol dictionary (including symbol and argument extension records). Setting all bits to zero in *symbol_total* means that the symbol dictionary is not defined in a SOM. *Symbol_total* must have a value in the range 0 to $2^{31}-1$.

fixup_request_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the fixup request dictionary. Setting all bits to zero in *fixup_request_location* indicates that the fixup request array is not defined in a SOM. The fixup request array must start on a word boundary. *Fixup_request_location* must have a value in the range 0 to $2^{31}-1$.

fixup_request_total

This field contains the number of fixup request records in the fixup request dictionary. Setting all bits to zero in *fixup_request_total* means that the fixup request dictionary is not defined in a SOM. *fixup_request_total* must have a value in the range 0 to $2^{31}-1$.

symbol_strings_location

Symbol_strings_location is a pointer to an area that contains symbol names and compilation unit names. It is a byte offset relative to the first byte of the SOM header. Setting all bits to zero in *symbol_strings_location* indicates that there are no symbol or compilation unit names in a SOM. The symbol string area must start on a word boundary. *Symbol_strings_location* must have a value in the range 0 to $2^{31}-1$.

symbol_strings_size

This field contains the byte length of the symbol dictionary string area. Setting all bits to zero in *symbol_strings_size* indicates that the symbol string area is not defined in a SOM. *Symbol_strings_size* must be a multiple of 4 bytes and be in the range 0 to $2^{31}-1$.

unloadable_sp_location

This is a byte offset relative to the first byte of the SOM header that points to the first byte of the data for unloadable spaces. Setting all bits to zero in *unloadable_sp_location* indicates that there are no unloadable spaces defined in a

SOM. The data for unloadable spaces must be double-word aligned. *Unloadable_sp_location* must have a value in the range 0 to $2^{31}-1$.

unloadable_sp_size

This field contains the byte length of the data for unloadable spaces. Setting all bits to zero in *unloadable_sp_size* indicates that the data for unloadable spaces is not defined in a SOM. *Unloadable_sp_size* must be a multiple of 8 bytes and be in the range 0 to $2^{31}-1$.

checksum

This field is the exclusive OR of all the words, excluding the checksum field, of the SOM header. It will be used to quickly evaluate valid SOM headers.

3.2 Compilation Unit Records

A compilation unit is defined as the set of procedures compiled by a single invocation of a given compiler. The compilation unit dictionary contains one entry for each SOM created by an invocation of a compiler. The Compilation Unit Record contains information for version identification of the compiler which generated the given SOM. Each entry contains information which may be used to identify the source name, the compiler language, the compiler product number, and the particular version of the compiler used. Lastly, each entry contains time stamps which identify the last modification made to the (main) source file and the time of compilation.

```
struct compilation_unit {  
    union name_p    name;  
    union name_pt   language_name;  
    union name_pt   product_id;  
    union name_pt   version_id;  
    unsigned int     reserved : 31;  
    unsigned int     chunk_flag : 1;  
    struct sys_clock compile_time;  
    struct sys_clock source_time;  
};
```

Figure 2-10: Definition of Compilation Unit Dictionary Record

name

This field contains a byte offset relative to the symbol string area which points to the first character of the string defining the entry name. The compilers should supply the name of the source file that produced the SOM.

language_name

This field contains a 32-bit index into the symbol string area, which points to the first character of the name of the language used when creating this SOM.

product_id

This field contains a 32-bit index into the symbol strings area which points to the first character of the identification number of the compiler.

version_id

This field contains a 32-bit index into the symbol strings area which points to the first character of the version id of the compiler.

reserved

These bits are reserved for future expansion.

chunk_flag

This field indicates that the compilation unit is not the first SOM in a multiple chunk compilation.

compile_time

compile time is a 64 bit value that represents the time the file was last compiled. The file time is actually composed of two 32 bit quantities where the first 32 bits is the number of seconds that have elapsed since January 1, 1970 (at 0:00 GMT), and the second 32 bits is the nano second of the second (which requires 30 bits to represent).

This value is independent of any modification time maintained by other subsystems (e.g. the file system). The use of this field is optional, but if it is not used it will be set to zero.

source_time

The file time is a 64 bit value that represents the time the file was last modified. The time is represented in the same format as *compile_time*.

This value is independent of any modification time maintained by other subsystems (e.g. the file system). The use of this field is optional, but if it is not used it will be set to zero.

3.3 Space Dictionary

The space dictionary consists of a collection of space records in contiguous bytes in the file. A space record is a template which defines attributes of a space (which correspond to the address spaces defined in the PA-RISC Architecture). Spaces, in general, are used as logical divisions of virtual memory. Current implementations may allow only one code and one data space. The *access_control_bits* field of a subspace record indicate whether a subspace is code or data. Each space record will indicate the space name, a pointer to the start of the subspace list, and a pointer to the start of the list of data initialization pointers that are to be applied to a space.

```
struct space_dictionary_record {
    union name_pt    name;           /* index to subspace name */
    unsigned int     is_loadable : 1; /* space is loadable */
    unsigned int     is_defined : 1;  /* space is defined within file */
    unsigned int     is_private : 1;  /* space is not sharable */
    unsigned int     has_intermediate_code: 1; /* contain intermediate code */
    unsigned int     is_tspecific : 1; /* is thread specific */
    unsigned int     reserved : 11;   /* reserved for future expansion */
    unsigned int     sort_key : 8;    /* sort key for space */
    unsigned int     reserved2 : 8;    /* reserved for future expansion */
    int              space_number;     /* space index */
    int              subspace_index;   /* index into subspace dictionary */
    unsigned int     subspace_quantity; /* number of subspaces in space */
    int              loader_fix_index; /* loader usage */
    unsigned int     loader_fix_quantity; /* loader usage */
    int              init_pointer_index; /* index into data(initialization)
                                         pointer array */
    unsigned int     init_pointer_quantity; /* number of data (init) pointers */
};
```

Figure 2-11: Space Dictionary Record Definition

name_pt

The field *name_pt* is an index into the space string area which points to the first character of the ascii representation of the space name. The index is a byte offset relative to the *space_strings_location* field of the SOM header. See the section on string areas for more details on the format of a name. *name_pt* is a byte offset relative to the field *space_strings_location* in the SOM header. *name_pt* can be converted to a file byte offset by:

offset = *name_pt*

+ *space_strings_location* (found in the SOM header)

+ address of the first byte of the SOM header.

If *name_pt* is greater than the field *space_strings_size* in the SOM header it is an error. Setting all bits to zero in *name_pt* indicates a null name pointer. *name_pt* must have a value in the range 0 to $2^{31}-1$.

is_loadable

Bit 0

If a space is loadable this flag is set to one. If a space is not loadable this flag is set to zero. Code and data for a load module will be the typical loadable spaces.

is_defined

Bit 1

If a space is defined in the file in which the space record resides the flag is set to one. If a space is not defined in the file in which the space record resides then the flag is set to zero.

is_private

Bit 2

If this flag is set then the space is non-sharable.

has_intermediate_code

Bit 3

This bit indicates that the space has only intermediate code in it (ISOM). The space will also be marked unloadable at the same time. The symbol dictionary information is complete but not meaningful since the \$TEXT\$ and \$PRIVATE\$ spaces are empty.

is_tspecific

Bit 4

\$TSPECIFIC\$ (TSD) space.

Reserved

Bit 5-15

These bits are reserved for future expansion.

sort_key

Bits 16-23

This field specifies a sort key which may be used by the linker for ordering spaces in the output file.

reserved1

Bits 24-31

These bits are reserved for future expansion.

space_number

This field specifies the number assigned to this space. Current implementations may default the space number values. Current implementations may ignore this field.

subspace_index

This field is an index into the subspace dictionary. All of the subspace records for a particular space will be in contiguous records in the subspace dictionary.

subspace_index can be converted to a file byte offset by:

offset = *subspace_index* * size of (subspace record) +

subspace_dictionary_location (found in the SOM header)

+ address of the first byte of the SOM header.

If *subspace_index* is greater than the field *subspace_dictionary_total* in the SOM header it is an error. If *subspace_index* is negative then there are no subspaces defined for that space. *Subspace_index* must have a value in the range -2^{31} to $2^{31}-1$.

subspace_quantity

Subspace_quantity is a number indicating how many subspaces are in a space. If *subspace_index* + *subspace_quantity* is greater than the field *subspace_dictionary_total* in the SOM header it is an error. If

subspace_quantity is zero then there are no subspaces in that space. *Subspace_quantity* must have a value in the range 0 to $2^{31}-1$.

loader_fix_index

Index into loader fixup array.

loader_fix_quantity

Number of loader fixups in this space.

init_pointer_index

This field is an index into the initialization pointer array. All of the initialization pointers for a particular space will be in contiguous records in the initialization pointer array. *init_pointer_index* can be converted to a file byte offset by:

offset = *init_pointer_index* * sizeof (initialization pointer record)
+ *init_array_location* (found in the SOM header)
+ address of the first byte of the SOM header.

If *init_pointer_index* is greater than the field *init_array_total* in the SOM header it is an error. If *init_pointer_index* is negative then there are no initialization pointers for that space. *Init_pointer_index* must have a value in the range -2^{31} to $2^{31}-1$.

init_pointer_quantity

Init_pointer_quantity is a number indicating how many initialization pointers are in the space. If *init_pointer_index* + *init_pointer_quantity* is greater than *init_array_total* in the SOM header it is an error. If *init_pointer_quantity* is zero then there are no initialization pointers for that space. *Init_pointer_quantity* must have a value in the range 0 to $2^{31}-1$.

3.4 Subspace Dictionary

A subspace corresponds to a logical subdivision of an address space. A subspace record is a template used to define the attributes of a subspace. The subspace dictionary consists of a collection of subspace records in contiguous bytes in the file. The subspace records are grouped by space. They contain information that can be used for relocation, setting of access rights of pages, determining how to build data areas, requesting a subspace to be locked in memory, and alignment requests.

Subspaces cannot be broken up into smaller entities, therefore there must not be any inter-subspace references generated without also generating a fixup for that reference.

Compilers are responsible for insuring that all branches can reach the beginning of their subspace.

```

struct subspace_dictionary_record {
    int            space_index;
    unsigned int   access_control_bits :7; /* access for PDIR entries */
    unsigned int   memory_resident :1; /* lock in memory during
                                         execution */
    unsigned int   dup_common :1; /* data name clashes allowed */
    unsigned int   is_common :1; /* subspace is a common
block*/
    unsigned int   is_loadable :1;
    unsigned int   quadrant :2; /* quadrant request */
    unsigned int   initially_frozen :1; /* must be locked into memory
                                         when OS is booted */
    unsigned int   is_first :1; /* must be first subspace */
    unsigned int   code_only :1; /* must contain only code */
    unsigned int   sort_key :8; /* subspace sort key */
    unsigned int   replicate_init :1; /* init values replicated to
                                         fill subspace_length */
    unsigned int   continuation :1; /* subspace is a continuation*/
    unsigned int   is_tspecific :1; /* Is thread specific ?*/
    unsigned int   is_comdat :1; /* Is for COMDAT subspaces?*/
    unsigned int   reserved :4;
    int            file_loc_init_value; /* file location or
                                         initialization value */
    unsigned int   initialization_length;
    unsigned int   subspace_start; /* starting offset */
    unsigned int   subspace_length; /* number of bytes defined
                                         by this subspace */
    unsigned int   reserved2 :5;
    unsigned int   alignment :27; /* alignment required for the
                                         subspace (largest alignment
                                         requested for any item in
                                         the subspace) */
    union name_pt   name; /* index of subspace name */
    int            fixup_request_index; /* index into fixup array */
    unsigned int   fixup_request_quantity; /* number of fixup requests */
};

```

Figure 2-12: Subspace Dictionary Record Definition

space_index

This field is a index into the space dictionary. All of the space records will be in contiguous records in the space dictionary. *space_index* can be converted to a file byte offset by:

$$\begin{aligned} \text{offset} = & \text{space_index} * \text{size of (space record)} \\ & + \text{space_dictionary_location (found in the SOM header)} \\ & + \text{address of the first byte of the SOM header.} \end{aligned}$$

If a *space_index* is greater than the field *space_quantity* in the SOM header record it is an error. If *space_index* is negative it is an error. *Space_index* must have a value in the range 0 to $2^{31}-1$.

access_control_bits

The *access_control* bits specify the access rights and privilege level of the subspace. They also specify whether the subspace contains code or data. Bits 0-7 of the *access_control_bits* are defined in Table 11.

TABLE 11
Subspace Access Control Bits

Type (3 bits)	Read/Write/Execute/Gateway (4 bits)		
	1st Field (PL1)	2nd Field (PL2)	Usage
0	Read	Not Used	Read only data page
1	Read	Write	Normal data page
2	Read/Xleast	Xmost	Normal code page
3	Read/Xleast	Write/Xmost	Dynamic code page
4	Xleast	Xmost	Gateway to PL0
5	Xleast	Xmost	Gateway to PL1
6	Xleast	Xmost	Gateway to PL2
7	Xleast	Xmost	Gateway to PL3

memory_resident

If this flag is set to one then the subspace is to be locked in physical memory once the subspace goes into execution.

dup_common

Bit 8

If this flag is set, then there may be more than one universal data symbol of the same name and the linker will not give a duplicate definition type of error. This field is used to facilitate implementation of Fortran initialized common and Cobol common.

is_common

Bit 9

This flag is set to one if the subspace is to define an initialized common data block. For example, Fortran initialized common, and Cobol common data blocks. Only one initialized data block is allowed per *is_common* subspace.

is_loadable

Bit 10

This flag is set to 1 if a subspace is loadable. Loadable subspaces must reside in loadable spaces. Unloadable subspaces must reside in unloadable spaces.

quadrant

Bits 11-12

This is to specify which of the four possible quadrants of a space that this subspace is going to reside. Current implementations may ignore this field, and place the subspace in a pre-determined quadrant.

TABLE 12

Quadrant Values

Bits	Meaning
00	Quadrant 0
01	Quadrant 1
10	Quadrant 2
11	Quadrant 3

initially_frozen

Bit 13

If this flag is set to one then the subspace is to be locked in physical memory when the operating system is being booted.

is_first

Bit 14

If this flag is set then the subspace must be the first subspace.

code_only

Bit 15

If set, this flag specifies that this subspace must only contain code (no literal data).

sort_key

Bits 16-23

This field contains the primary sort key by which the linker arranges subspaces in an output file. Subspaces are first ordered by the sort key, then are arranged according to the subspace name. Within sort keys, the linker groups subspaces by their name but it does not sort by name. Instead, the subspaces are output in the order in which the linker first encounters each name.

replicate_init

Bit 24

If the initialization contained in the file is shorter than the subspace length, replicate it to fill the length of the subspace.

continuation

Bit 25

If set, the subspace is a continuation of a previous subspace and the two (or more) subspaces should be treated as a single unit.

is_tspecific

Bit 26

Thread specific subspace.

is_comdat

Bit 27

Thread specific subspace.

reserved

Bits 28-31

These bits are reserved for future use.

file_loc_init_value

If *initialization_length* field is non-zero, the subspace is initialized, and this field contains a byte offset relative to the first byte of the SOM header. In other words, it is a file location of the initialization image.

If *initialization_length* is zero then this field contains a 32 bit quantity which is used as an initialization pattern for the entire subspace. The total length of the subspace is defined by the *subspace_length* field. This is how BSS subspaces are represented.

initialization_length

This field contains the size in bytes of the initialization area in the file. If this field is zero then the value contained in the field *file_loc_init_value* is used as the initialization pattern for the subspace.

The *initialization_length* field can also be non-zero, but less than the *subspace_length* field. In this case, the length of the initialization image is given by *initialization_length*, and the remainder of the subspace, up to *subspace_length*, is initialized with zeros.

subspace_start

This is a byte address of where the subspace is to start relative to the beginning of a space. It is a virtual address that indicates the assumed beginning of that subspace in memory. This value in conjunction with *subspace_length* will be used to insure that subspaces do not overlap. *Subspace_start* must have a value in the range 0 to $2^{32}-1$.

subspace_length

This is the length in bytes of a subspace. A total length of a space will be kept, and if the addition of all of the *subspace_length* fields in a space is greater than $2^{32}-1$ then it is an error.

reserved2

Bits 0-15

These bits are reserved for future expansion.

alignment

Bits 16-31

This 2-byte field specifies what alignment is required for the subspace. The alignment request is in bytes. The subspace will start on the alignment byte boundary. The alignment value must be greater than zero.

name

The field *name* is an index into the space/subspace string area. The index is a byte relative offset which points to the first character of the string. See the chapter on string areas for more details on the format of a name. *name* can be converted to a file byte offset by:

offset = *name* + *space_strings_location* (found in the SOM header)

+ address of the first byte of the SOM header.

If *name* is greater than the field *space_strings_size* in the SOM header it is an error. Setting the field name to zero means that it is a null name pointer.

fixup_request_index

This field is an index into the fixup request array. All of the fixup request records for a particular subspace will be in contiguous records in the fixup request array.

fixup_request_index can be converted to a file byte offset by:

offset = *fixup_request_index* * size of (fixup record)

+ *fixup_request_location* (found in the SOM header)

+ address of the first byte of the SOM header.

If *fixup_request_index* is greater than the field *fixup_request_total* in the SOM header record it is an error. If *fixup_request_index* is negative then there are no fixup requests for that subspace. *Fixup_request_index* must have a value in the range -2^{31} to $2^{31}-1$.

fixup_request_quantity

Fixup_request_quantity is a number indicating how many fixup requests there are for a subspace. If *fixup_request_index* + *fixup_request_quantity* is greater than the field *fixup_request_total* in the SOM header record it is an error.

Fixup_request_quantity must have a value in the range 0 to $2^{31}-1$. If

fixup_request_quantity is zero then there are no fixup requests for that subspace.

3.5 String Areas

The string area contains all symbols used in the SOM, including space names, subspace names, export names, import requests, and compilation unit names. There will be two string areas; one for space and subspace names, and one for symbols and compilation unit names.

The first word of each string contains the total number of characters in the string. The byte immediately following the last byte of the string will be zero (the null character). Successive strings will begin on the next word boundary.

string header

This field contains the total number of characters contained in the string (does not include the terminating null character).

string data

Bits 0-??

The string is defined by the character data given here.

3.6 Fixup Requests

In the object files, relocation entries consist of a stream of bytes. The *fixup_request_index* field in the subspace dictionary entry is a byte offset into the fixup dictionary defined by the file header, and the *fixup_request_quantity* field defines the length of the fixup request stream, in bytes, for that subspace. The first byte of each fixup request (the opcode) identifies the request and determines the length of the request.

In general, the fixup stream is a series of linker instructions that governs how the linker places data in the a.out file. Fixups requests can be grouped into the following five categories:

- fixup requests that cause the linker to copy one or more bytes from the input subspace to the output subspace without change. For example, the R_NO_RELOCATION fixup that cause the linker to copy n bytes to the output subspace with no relocation.
- fixups that direct the linker to relocate words or resolve external references. For example, the R_DP_RELATIVE fixup used to relocate the target symbol in the output subspace. The address is calculated based on the offset from \$global\$, the data pointer (r27).

- fixups that direct the linker to insert zeroes in the output subspace. For example, the `R_REPEATED_INIT` to replicate the data to fill `n` bytes of initialized value in the output subspace.
- fixups that direct the linker to leave areas uninitialized without copying any data from the input subspace. For example, the `R_UNINIT` fixup that tells the linker to skip bytes in the output subspace.
- fixups that describe points in the code without contributing any new data to the output file. These fixups **DO** indirectly affect the output, they are considered fixups for changing the environment. For example, the rounding mode fixups (`R_N_MODE`, `R_D_MODE` and the `R_ENTRY` and `R_EXIT` fixups). They do affect how the data are to be interpreted for the output file.

When applying the fixups, the linker examines the instruction opcode (the high-order six bits of the instruction) to determine the format of the instruction and what part of the word should be relocated. The linker also selects a default field selector based on the opcode if an explicit field selector override is not in effect.

In earlier phases of the link, however, the instruction opcode is not available for examination (the linker does not read the contents of a subspace until the final link phase). The linker must therefore be able to infer certain opcodes and field selectors from the fixup information alone. In particular, each `R_PCREL_CALL` fixup is assumed to apply to a branch-and-link (BL) instruction with a (default) `F%` field selector, unless the fixup is preceded by an `R_LSEL` or `R_RSEL` override. For `R_PCREL_CALL` fixups not preceded by either of these overrides, the linker will test the branch distance to determine if a long branch stub is required, and will insert a long branch stub if necessary. Thus, it is necessary that long-format pc-relative calls (using the `ADDIL` and `LDO` instructions) use explicit overrides, as in this example (from Section 2.5.5):

```
BL    .+8, rp
ADDIL L'target-$L0+4, rp
LDO   R'target-$L1+8(r1), r1
$L0:  LDSID (r1), r31
$L1:  MTSP  r31, sr0
      BLE  0(sr0, r1)
      COPY r31, rp
```

Here, the `ADDIL` should be tagged with `R_LSEL` and `R_PCREL_CALL` fixups, and the `LDO` should be tagged with `R_RSEL` and `R_PCREL_CALL` fixups.

3.6.1 Fixup Rounding Modes

TABLE 13

R and L-Class Fixups

L'	Set bits 21-31 to 0 (set the rightmost 11 bits to 0)
R'	Set bits 0-20 to 0
LD'	Add 0x800, set bits 21-31 to 0
RD'	Set bits 0-20 to 1
LR'	Round constant before evaluating expression, set bits 21-31 to 0
RR'	Round constant before evaluating expression, set bits 0-20 to 0, add (constant - round(constant)) round(constant) = (constant + 0x1000) & ~0x1FFF
LS'	If (bit 21) then add 0x800 and set bits 21-31 to 0
RS'	Sign extend from bit 21

All direct and dp-relative effective address calculations use the LR and RR rounding modes. In these rounding modes, the left part is computed based on a rounded constant instead of the actual constant. The constant is rounded to the nearest multiple of 8192 prior to computing the effective address. The right part is computed as the difference between the full value of the expression and the value used in the left-part relocation. Because the difference between the original constant and the rounded constant can be no larger than 4K, this result will always fit in a signed 14-bit field. This permits several load and store instructions to reuse the result of a single ADDIL or LDIL instruction, as long as the symbol index and the rounded value of the constant are identical.

For pc-relative relocations, the standard L and R rounding modes are used. The expression is computed based on the actual effective address.

The following C language functions define the operation of the LR, RR, L, R, and RND functions:

```
unsigned long LR(unsigned long x, unsigned long constant)
```

```
{
    return L(x + RND(constant));
}
```

```
unsigned long RR(unsigned long x, unsigned long constant)
```

```
{
    return R(x + RND(constant)) + (constant - RND(constant));
}
```

```
}  
  
unsigned long L(unsigned long x)  
  
{  
  
    return (x & 0xffff800);  
  
}
```

```
unsigned long R(unsigned long x)  
  
{  
  
    return (x & 0x000007ff);  
  
}
```

```
unsigned long RND(unsigned long x)  
  
{  
  
    return ((x + 0x1000) & 0xffffe000);  
  
}
```

3.6.2 Interpretation of rounding mode and field selector

In a relocatable object file, the immediate fields of the instructions contain only the constant part of the expression. For a “symbol+constant” expression, the `R_CODE_ONE_SYMBOL` fixup identifies the symbol and the immediate field contains the constant. Whether the instruction is forming the left part or the right part of an address, the immediate field still contains the entire constant. If the constant is too large for the immediate field, the compilers precede the fixup with an `R_DATA_OVERRIDE` fixup that supplies the full 32 bits.

For a “symbol-\$global\$+constant” expression, the `R_DP_RELATIVE` fixup identifies the symbol, and the constant is in the immediate field. For other kinds of expression, the linker resort to the more general stack-based expression evaluation mechanism, and the `R_CODE_EXPR` fixup would be used. For example, “symbol1-symbol2+constant” would be represented by the fixup stream:

```
R_PUSH_SYM symbol1
```

```
R_PUSH_SYM symbol2
```

R_COMP1 R_SUB

R_CODE_EXPR

Again, the constant part of the expression is in the instruction itself.

The field selector in the assembly syntax really consists of two parts: the rounding mode (normal, D, R, S), and the field selector itself (F, L, or R).

The field selector is normally implied by the opcode. ADDIL and LDIL instructions only imply the L% field selector. Most other opcodes that take displacements or offsets imply the R% field selector, with the exception of the BL opcode, which implies the F% field selector, since it is most often used for a single-instruction procedure call. If the full 32-bit displacement for a BL instruction is too large (computed by the R_PCREL_CALL fixup), the linker creates a long branch stub. In all other cases, a displacement that does not fit in the instruction causes a link-time error.

Note that the default field selector in assembly syntax is F%. If an assembly instruction is coded with F% or with no field selector, the assembler must generate a field selector override fixup immediately preceding the regular fixup for that instruction. Likewise, if an assembly instruction with a field selector other than the default, an override must be generated.

The particular rounding mode selected in the assembly instruction affects only the fixups generated. If the rounding mode selected is different from the “current” rounding mode, a fixup is generated to change the current mode. Unlike the field selector override, the rounding mode is persistent, and must be changed back for the next instruction that uses a different mode.

The field selectors come in pairs, and an LDIL or ADDIL instruction must always be paired with an LDO/LDW/STW that uses the same rounding mode. The relations

$$L\%expr + R\%expr = expr$$
$$LD\%expr + RD\%expr = expr$$
$$LS\%expr + RS\%expr = expr$$
$$LR\%expr + RR\%expr = expr$$

always hold. R%expr is always positive, which implies that L%expr is always the first 2K boundary less than or equal to expr. This is the most straightforward definition.

RD%expr, however, is always negative, implying that LD%expr is always the first 2K boundary greater than expr. This mode is useful when the code is near a quadrant boundary, and the base register formed by LDIL and used by the LDW is on the higher quadrant. The space register used in an LDW instruction when the s field is 0 is determined solely by the upper two bits of the base register (not by the effective address). Therefore, if code is generated to access a non-zero based array, for example, this mode

can be used to ensure that the intermediate address is not down in quadrant 0 when the data to be accessed is in quadrant 1.

The next mode, **LS%/RS%**, is defined such that **RS%expr** is between -1024 and +1023, inclusive. This implies that **LS%expr** is the nearest 2K boundary to **expr**. If the second instruction of a pair is an **ADDI** instruction (or **SUBI**, **COMICLR**, ..etc.), this mode is essential, since there are only 11 bits of immediate field available, and the immediate is sign-extended.

The last mode, **LR%/RR%**, is the only one where the constant field is treated separately from the rest of the expression. This pair is defined like **L%** and **R%**, except that the lower bits of the constant do not participate in the **LR%** determination; they get added back in to the **R%** value. This allows the final value of **RR%expr** to be larger than 2K, but never too large to fit in the 14-bit signed immediate field of an **LDW**-class instruction. This mode is conveniently defined so that the compiler can share a single **ADDIL** instruction among several **LDW**-class instructions where the expressions are the same except for the constant part of the expression. Note that they can be shared as long as the constants are all equal in their upper bits. In other words, if the compiler knows that **LR%symbol+con1** will evaluate to the same thing as **LR%symbol+con2**, it can share one **ADDIL** instruction with both corresponding **LDW/STW** instructions using **RR%symbol+con1** and **RR%symbol+con2**. This is efficient when generating code to access structures and static data where several adjacent memory locations are all addressed by a single symbol.

3.6.3 Examples of applying the rounding mode

The following is an example of how the rounding modes are applied:

symbol 4 = 0x4000fff0

ADDIL 0x1000000,27 /* immediate is 8192 in decimal */

LDO 4104 (1), 25 /* immediate is in decimal as is */

For the **LR%** and **RR%** modes, the constant is rounded to the nearest 8K multiple before splitting the value in half. Then, after splitting, the difference is added back in to the right half. The following is the pseudo code for this algorithm:

```
#define FIXUP_ROUND(c) (((c) + 0x1000) & ~0x1fff)
```

```
expr = symbol_value + FIXUP_ROUND(constant);
```

```
left = expr & 0xFFFFF800;
```

```
right = (expr & 0x7FF) + (constant - FIXUP_ROUND(constant));
```

In this example, the **ADDIL** gets a “rounded” expression value of 0x4000fff0 + 0x1000 = 0x40010ff0, which gets truncated to 0x40010800. The **LDO** gets a “rounded” expres-

sion value of $0x4000fff0 + 0x1000 = 0x40010ff0$, which gets truncated to $0x7f0$, to which we add the difference between the constant $0x1008$ and the rounded constant ($0x1000$), resulting in $0x07f8$.

Thus, the ADDIL/LDO form the address $0x40010800 + 0x07f8 = 0x40010ff8$, which is the same as $0x4000fff0 + 0x1008$ (symbol #4 plus 4104).

As mentioned briefly in the previous section, the reason that this is done this way is that a single ADDIL can be accessed with many LDO/LDW/STW instruction, each of which may have a slightly different constant. As long as the constants all round to the same value, we can use a common ADDIL instruction for all of them. In practice, this works for accesses to the fields of a structure, where we use the same symbol with different displacements in several loads or stores.

3.6.4 Apply Fixups on instructions

The linker apply fixups to instructions in the following three steps:

1. Calculate the effective address. This depends on the fixup type. This usually involves checking the opcode and extracting a constant from the immediate or displacement field. Step one is where the linker actually looks at the opcode, decide what the default field selector should for step two, and identify which of the six instruction formats to use in step three.
2. Apply the field selector. This can be implied by the opcode, or can be overridden by a field selector override fixup (R_xSEL). It also depends on the current rounding mode. The default field selector is L% for LDIL and ADDIL, F% for BL, COMB, ADDDB, and BB family of opcodes, and R% for everything else. For example, BE/BLE instructions have an implicit R% field selector. If one were to code a BE/BLE in assembler without the R%, an F% field selector override fixup (R_FSEL) for that instruction is needed. This tells the linker not to chop off the top 21 bits of the effective address, and try to fit the address into the instruction as is. If the address is too large, the linker would issue a diagnostic such as “displacement too large”. Step two is where the expression gets converted from an absolute address to a pc-relative address. If the instruction format is i_rel12 or i_rel17, (pc+8) is subtracted from the effective address to obtain the proper pc-relative displacement.
3. Apply the resulting value to the target instruction. The opcode determines the actual disposition of the various bits. There are really only six different instruction formats: i_exp11 (ADDI), i_exp14 (LDW), i_exp21 (LDIL), i_rel12 (ADDIB), i_rel17 (BL) and i_abs17 (BE). These names are in <reloc.h>. i_rel17 and i_abs17 are really the same instruction format, but the linker adjust the space register field in the BE-class instructions based on where the target is.

3.6.5 List of fixup requests

The meaning of each fixup request is described below. The opcode ranges and parameters for each fixup are described in the table further below.

TABLE 14

Fixup Requests

R_NO_RELOCATION	Copy L bytes with no relocation.
R_ZEROES	Insert L zero bytes into the output subspace.
R_UNINIT	Skip L bytes in the output subspace.
R_RELOCATION	Copy one data word with relocation. The word is assumed to contain a 32-bit pointer relative to its own subspace. It describes a single word whose value must be relocated, assuming it contains an address constant of a location within the same subspace. The word to be relocated comes from the initialization image, not from the fixup stream.
R_DATA_ONE_SYMBOL	Copy one data word with relocation relative to an external symbol whose symbol index is S.
R_DATA_PLABEL	Copy one data word as a 32-bit procedure label, referring to the symbol S. The original contents of the word should be 0 (no static link) or 2 (static link required).
R_SPACE_REF	Copy one data word as a space reference. This fixup request is not currently supported.
R_REPEATED_INIT	Copy L bytes from the input subspace, replicating the data to fill M bytes in the output subspace.
R_PCREL_CALL	Copy one instruction word with relocation. This word is assumed to be a pc-relative procedure call using the branch-and-link instruction (BL), unless an R_LSEL or R_RSEL override is in effect. The target procedure is identified by symbol S, and the parameter relocation bits are R. Note: The displacement on pc-relative calls is assumed to be -8, unless preceded by a data override fixup.
R_SHORT_PCREL_MODE	this specifies that any following R_PCREL_CALL fixup (with the default field selector) is applied to a BL instruction with a maximum 17-bit signed displacement. It is a single-byte mode change fixup, and is the initial default mode.
R_LONG_PCREL_MODE	this specifies that any following R_PCREL_CALL fixup (with the default field selector) is applied to a BL instruction with a maximum 22-bit signed displacement (i.e., a BLL instruction). It is a single-byte mode change fixup.
R_ABS_CALL	Copy one instruction word with relocation. The word is assumed to be an absolute procedure call instruction (for example, BLE). The target procedure is identified by symbol S, and the parameter relocation bits are R. Note: absolute calls using LDIL/ADDIL and BE/BLE always default to the R'/L' rounding mode, unless explicitly preceded by a rounding mode override (this is the one exception to the rounding mode being a persistent change). The displacement on absolute calls is assumed to be 0, unless preceded by a data override fixup.

TABLE 14

Fixup Requests

R_DP_RELATIVE	Copy one instruction word with relocation. The word is assumed to be a dp-relative load or store instruction (for example, ADDIL, LDW, STW). The target symbol is identified by symbol S. The linker forms the difference between the value of the symbol S and the value of the symbol \$global\$. By convention, the value of \$global\$ is always contained in register 27. Instructions may have a small constant in the displacement field of the instruction.
R_DATA_GPREL	When not building a shared library, the linker forms the difference between the value of the symbol S and the value of the symbol \$global\$. When building a shared library, the linker computes a linkage table offset relative to register 19 (reserved for a linkage table pointer in position-independent-code) for the symbol S.
R_INDIRECT_CALL	Directs the linker to substitute calls to \$\$dyncall with calls to \$\$dyncall_external().
R_PLT_REL	This is analogous to R_DLT_REL; it requests the displacement field of the instruction to be filled with the value <linkage table pointer address - PLT slot for symbol>. It is used for instructions in inlined import stubs. It is only available in a 4-byte form, in which the symbol index is encoded in the last 3 bytes of the fixup.
R_DLT_REL	Copy one instruction word with relocation. The word is assumed to be a register r19-relative load or store instruction (for example, LDW, LDO, STW). The target symbol is identified by symbol S. The linker computes a linkage table offset relative to register 19 (reserved for a linkage table pointer in position-independent-code) for the symbol S.
R_CODE_ONE_SYMBOL	Copy one instruction word with relocation. The word is assumed to be an instruction referring to symbol S (for example, LDIL, LDW, BE).
R_MILLI_REL	Copy one instruction word with relocation. The word is assumed to be a short millicode call instruction (for example, BLE). The linker forms the difference between the value of the target symbol S and the value of symbol 1 in the module's symbol table. By convention, the value of symbol 1 should have been previously loaded into the base register used in the BLE instruction. The instruction may have a small constant in the displacement field of the instruction.
R_CODE_PLABEL	Copy one instruction word with relocation. The word is assumed to be part of a code sequence forming a procedure label (for example, LDIL, LDO), referring to symbol S. The LDO instruction should contain the value 0 (no static link) or 2 (static link required) in its displacement field.
R_BREAKPOINT	Copy one instruction word conditionally. On HP-UX, the linker always replaces the word with a NOP instruction.
R_ENTRY	Define a procedure entry point. The stack unwind bits, U, and the frame size, F, are recorded in a stack unwind descriptor (copied to words 3 and 4 for the unwind region).

TABLE 14

Fixup Requests

R_ALT_ENTRY	Define an alternate procedure entry point.
R_EXIT	Define a procedure exit point.
R_BEGIN_TRY	<p>Define the beginning of a try/recover region.</p> <p>The try/recover mechanism is designed to support features such as try/recover in Pascal and try/catch in C++. The recover table is constructed by the linker and consists of some number of recover descriptors. A recover descriptor consists of three words:</p> <p>word 1: the starting address of the “try” region.</p> <p>word 2: the ending address of the “try” region.</p> <p>word 3: an address pointing at language-dependent region.</p> <p>For example:</p> <p>Pascal: the address of the exception handler</p> <p>Ada: the address of a descriptor block</p> <p>C++: a pointer to a C++ data structure</p> <p>The linker builds try/recover descriptors based on the R_BEGIN_TRY/R_END_TRY fixups. The first two words of the try/recover descriptor are just the addresses of the beginning and end of the guarded region as indicated by the placement of the fixups. The third word is the address of the end of the guarded region (the second word) plus four times the argument of R_END_TRY. This region is sometimes referred to as the “recover block”. The END_TRY fixup contains a pc-relative offset to the recover block. The actual meaning of the recover block is language dependent. In Pascal, it is just a pointer to the recover code, so it is often the address immediately following the guarded region so the constant in the END_TRY fixup is often 0.</p> <p>The C++ exception handling mechanism uses a recover block that points to other information. The first word of this recover block is a pointer to the code in the catch block. Like the Pascal case above, the catch block often immediately follows the guarded region, so this pointer often points back to the first instruction beyond the END_TRY fixup. Since this pointer is actually a self-relative offset, it often is the same number as was found in the END_TRY fixup. This may be a frequent case, but it is not guaranteed, for example, nested try/catch blocks will probably show a difference.</p>
R_END_TRY	Define the end of a try/recover region. The offset R defines the distance in words from the end of the region to the beginning of the recover block.
R_BEGIN_BRTAB	Define the beginning of a branch table.
R_END_BRTAB	Define the end of a branch table.
R_STATEMENT	Define the beginning of statement number N.

TABLE 14
Fixup Requests

R_DATA_EXPR	Pop one word from the expression stack and copy one data word from the input subspace to the output subspace, adding the popped value to it.
R_CODE_EXPR	Pop one word from the expression stack, and copy one instruction word from the input subspace to the output subspace, adding the popped value to the displacement field of the instruction.
R_FSEL	Use an F' field selector for the next fixup request instead of the default appropriate for the instruction. An F field selector denotes "no change". The "default" modes can be any of the R-class or L-class field selectors.
R_LSEL	Use an L'-class field selector for the next fixup request instead of the default appropriate for the instruction. Depending on the current rounding mode, L', LS', LD', or LR' may be used.
R_RSEL	Use an R-class field selector for the next fixup request instead of the default appropriate for the instruction. Depending on the current rounding mode, R', RS', RD', or RR' may be used.
R_N_MODE	Select round-down mode (L'/R'). This is the default mode at the beginning of each subspace. This setting remains in effect until explicitly changed or until the end of the subspace.
R_S_MODE	Select round-to-nearest-page mode (LS'/RS'). This setting remains in effect until explicitly changed or until the end of the subspace.
R_D_MODE	Select round-up mode (LD'/RD'). This setting remains in effect until explicitly changed or until the end of the subspace.
R_R_MODE	Select round-down-with-adjusted-constant mode (LR'/RR'). This setting remains in effect until explicitly changed or until the end of the subspace.
R_DATA_OVERRIDE	Use the constant V for the next fixup request in place of the constant from the data word or instruction in the input subspace.
R_TRANSLATED	Toggle "translated" mode. This fixup request is generated only by the linker during a relocatable link to indicate a subspace that was originally read from an old-format relocatable object file.
R_AUX_UNWIND	Define an auxiliary unwind table. CN is a symbol index of the symbol that labels the beginning of the compilation unit string table. SN is the offset, relative to the CN symbol, of the scope name string. SK is an integer specifying the scope kind.
R_COMP1	Stack operations. The second byte of this fixup request contains a secondary opcode. In the descriptions below, A refers to the top of the stack and B refers to the next item on the stack. All items on the stack are considered signed 32-bit integers.

TABLE 14

Fixup Requests

R_PUSH_PCON1	Push the (positive) constant V.
R_PUSH_DOT	Push the current virtual address.
R_MAX	Pop A and B, then push $\max(A, B)$.
R_MIN	Pop A and B, then push $\min(A, B)$.
R_ADD	Pop A and B, then push $A + B$.
R_SUB	Pop A and B, then push $B - A$.
R_MULT	Pop A and B, then push $A * B$.
R_DIV	Pop A and B, then push B / A .
R_MOD	Pop A and B, then push $B \% A$.
R_AND	Pop A and B, then push $A \& B$.
R_OR	Pop A and B, then push $A B$.
R_XOR	Pop A and B, then push $A \text{ XOR } B$.
R_NOT	Replace A with its complement.
R_LSHIFT	If $C = 0$, pop A and B, then push $B \ll A$. Otherwise, replace A with $A \ll C$.
R_ARITH_RSHIFT	If $C = 0$, pop A and B, then push $B \gg A$. Otherwise, replace A with $A \gg C$. The shifting is done with sign extension.
R_LOGIC_RSHIFT	If $C = 0$, pop A and B, then push $B \gg A$. Otherwise, replace A with $A \gg C$. The shifting is done with zero fill.
R_PUSH_NCON1	Push the (negative) constant V.
R_COMP2	More stack operations.
R_PUSH_PCON2	Push the (positive) constant V.
R_PUSH_SYM	Push the value of the symbol S.
R_PUSH_PLABEL	Push the value of a procedure label for symbol S. The static link bit is L.
R_PUSH_NCON2	Push the (negative) constant V.
R_COMP3	More stack operations.
R_PUSH_PROC	Push the value of the procedure entry point S. The parameter relocation bits are R.
R_PUSH_CONST	Push the constant V.
R_PREV_FIXUP	The linker keeps a queue of the last four unique multi-byte fixup requests; this is an abbreviation for a fixup request identical to one on the queue. The queue index X references one of the four; $X = 0$ refers to the most recent. As a side effect of this fixup request, the referenced fixup is moved to the front of the queue.
R_SEC_STMT	Secondary statement number.
R_NOSEL	Indicates that the following fixup is applied to the first of a three-instruction sequence to access data, generated by the compilers to enable the importing of shared library data.

TABLE 14
Fixup Requests

R_NISEL	Uses a N field selector for the next fixup request; this indicates that zero bits are to be used for the displacement on the instruction. This fixup is used to identify three-instruction sequences to access data (for importing shared library data).
R_LINETAB	<p>The compilers generate this fixup to request that debugging optimized code (DOC) line tables be built. The first parameter is a 1-byte version number which identifies the line table version (format). The actual value is not important to the linker. The second parameter is a symbol index to be used in conjunction with the third parameter, an offset, as a location which is to be filled with the offset (relative to the \$LINES\$ subspace) of the line table about to be built.</p> <p>The line number information is passed to the linker via the R_STATEMENT fixup request, which is embedded within the fixups for the code at statement boundaries. The R_STATEMENT fixup has three variants to handle one-, two-, and three-byte statement or line numbers as necessary. The actual meaning assigned to the number, whether it be statement number or line numbers, is irrelevant to the linker, and needs to be agreed upon only by the compiler and the end user of the line table information.</p>
R_LINETAB_ESC	<p>Fixup used to place escape entries into the line table. There are several escape entries defined in the line table format which are used by the debugger and other tools when processing the line table. Some of these escapes must be generated by the linker, the others are generated by the compiler and the linker does not need to know the details of these escapes. The escapes entries which are not generated by the linker are entered into the line table via a combination of the R_LINETAB_ESC and R_STATEMENT fixups.</p> <p>The second parm specifies how many of the following R_STATEMENT entries contain data to be entered directly into the line table (these statement fixups will not contain line numbers. Instead, they hold data which is to be placed directly into the line number table as part of an escape sequence. With the currently defined escapes the value of the second parameter will be in the range [0,4].</p>
R_LTP_OVERRIDE	<p>Override the following fixup which is expected to be an R_DATA_ONE_SYMBOL fixup. If the linker encounters an R_DATA_ONE_SYMBOL with the override set and it is building a shared library then it will convert the relocatable address representing the data item into a absolute offset by subtracting the presumed link time R19 value from the relocatable address. This will eventually require a run time relocation before it can be used to access the data item. There is currently no way to generate this fixup through the assembler interface.</p> <p>If the linker is not building a shared library, the absolute virtual address is placed in the target subspace.</p>

TABLE 14

Fixup Requests

R_COMMENT	Fixup used to pass comment information from the compiler to the linker. This fixup has a 5 byte argument that can be skipped and ignored by most applications.
R_TP_OVERRIDE	Override the following fixup which is expected to be an R_DP_RELATIVE, an R_DLT_REL, or an R_DATA_ONE_SYMBOL fixup. This override precede referencing TLS symbols.
R_RESERVED	Fixups in this range are reserved for internal use by the compilers and linker.

3.6.6 Fixup opcodes, lengths and parameters

The include file <reloc.h> defines constants for each major opcode. Many fixup requests use a range of opcodes; only a constant for the beginning of the range is defined.

Table 15 shows the mnemonic fixup request type and length and parameter information for each range of opcodes. In the parameters column, the symbol D refers to the difference between the opcode and the beginning of the range described by that table entry; the symbols B1, B2, B3, and B4 refer to the value of the next one, two, three, or four bytes of the fixup request, respectively.

TABLE 15

Fixup Request Opcodes (in hex) and Parameters

mnemonic	opcodes	length	parameters
R_NO_RELOCATION	0x00 - 17	1	$L = (D+1) * 4$
	0x18 - 1B	2	$L = (D < 8 + B1 + 1) * 4$
	0x1C-1E	3	$L = (D < 16 + B2 + 1) * 4$
	0x1F	4	$L = B3 + 1$
R_ZEROES	0x20	2	$L = (B1 + 1) * 4$
	0x21	4	$L = B3 + 1$
R_UNINIT	0x22	2	$L = (B1 + 1) * 4$
	0x23	4	$L = B3 + 1$
R_RELOCATION	0x24	1	none
R_DATA_ONE_SYMBOL	0x25	2	$S = B1$
	0x26	4	$S = B3$
R_DATA_PLABEL	0x27	2	$S = B1$
	0x28	4	$S = B3$
R_SPACE_REF	0x29	1	none
R_REPEATED_INIT	0x2A	2	$L = 4; M = (B1 + 1) * 4$
	0x2B	3	$L = (B1 + 1) * 4; M = (B1 + 1) * L$

TABLE 15

Fixup Request Opcodes (in hex) and Parameters

mnemonic	opcodes	length	parameters
	0x2C	5	$L = (B1 + 1) * 4; M = (B3 + 1) * 4$
	0x2D	8	$L = B3 + 1; M = B4 + 1$
	0x2E-2F		Reserved
R_PCREL_CALL	0x30 - 39	2	$R = \text{rbits1}(D); S = B1$
	0x3A-3B	3	$R = \text{rbits2}(D \ll 8 + B1); S = B1$
	0x3C-3D	5	$R = \text{rbits2}(D \ll 8 + B1); S = B3$
R_SHORT_PCREL_MODE	0x3E	1	mode shift to BL for R_PCREL_CALL(+DA2.0 only)
R_LONG_PCREL_MODE	0x3F	1	mode shift to BLL for R_PCREL_CALL (+DA2.0 only)
R_ABS_CALL	0x40-49	2	$R = \text{rbits1}(D); S = B1$
	0x4A-4B	3	$R = \text{rbits2}(D \ll 8 + B1); S = B1$
	0x4C-4D	5	$R = \text{rbits2}(D \ll 8 + B1); S = B3$
	0x4E-4F		Reserved
R_DP_RELATIVE	0x50-6F	1	$S = D$
	0x70	2	$S = B1$
	0x71	4	$S = B3$
R_DATA_GPREL	0x72	4	$S = B3$
	0x73-75		Reserved
R_INDIRECT_CALL	0x76	1	Specify target instruction is an indirect call through \$\$dyncall_external().
R_PLT_REL	0x77	4	Request the displacement field to be filled with the value (LPT addr - PLT slot for symbol).
R_DLT_REL	0x78	2	$S = B1$, DLT relative load/store
	0x79	4	$S = B3$
R_CODE_ONE_SYMBOL	0x80-9F	1	$S = D$
	0xA0	2	$S = B1$
	0xA1	4	$S = B3$
R_MILLI_REL	0xAE	2	$S = B1$
	0xAF	4	$S = B3$
R_CODE_PLABEL	0xB0	2	$S = B1$
	0xB1	4	$S = B3$
R_BREAKPOINT	0xB2	1	none
R_ENTRY	0xB3	9	$U, F = B8$ (U is 37 bits; F is 27 bits)
	0xB4	6	$U = B5 \gg 3; F = \text{pop A}$
R_ALT_ENTRY	0xB5	1	none
R_EXIT	0xB6	1	none

TABLE 15

Fixup Request Opcodes (in hex) and Parameters

mnemonic	opcodes	length	parameters
R_BEGIN_TRY	0xB7	1	none
R_END_TRY	0xB8	1	R = 0
	0xB9	2	R = B1 * 4
	0xBA	4	R = sign-extend(B3) * 4
R_BEGI_BRTAB	0xBB	1	none
R_END_BRTAB	0xBC	1	none
R_STATEMENT	0xBD	2	N = B1
	0xBE	3	N = B2
	0xBF	4	N = B3
R_DATA_EXPR	0xC0	1	none
R_CODE_EXPR	0xC1	1	none
R_FSEL	0xC2	1	none
R_LSEL	0xC3	1	none
R_RSEL	0xC4	1	none
R_N_MODE	0xC5	1	none
R_S_MODE	0xC6	1	none
R_D_MODE	0xC7	1	none
R_R_MODE	0xC8	1	none
R_DATA_OVERRIDE	0xC9	1	V = 0
	0xCA	2	V = sign-extend(B1)
	0xCB	3	V = sign-extend(B2)
	0xCC	4	V = sign-extend(B3)
	0xCD	5	V = B4
R_TRANSLATED	0xCE	1	none
R_AUX_UNWIND	0xCF	12	CU,SN,SK = B11 (CU is 24 bits;SN is 32)
R_COMP1	0xD0	2	OP = B1; V = OP & 0x3f; C = OP & 0x1f
R_COMP2	0xD1	5	OP = B1; S = B3; L = OP & 1; V = ((OP & 0x7f) << 24) S
R_COMP3	0xD2	6	OP = B1; V = B4; R = ((OP & 1) << 8) (V >> 16); S = V & 0xfffff
R_PREV_FIXUP	0xD3-D6	1	X = D
R_SEC_STMT	0xD7	1	none
R_N0SEL	0xD8	1	none
R_N1SEL	0xD9	1	none

TABLE 15

Fixup Request Opcodes (in hex) and Parameters

mnemonic	opcodes	length	parameters
R_LINETAB	0xDA	10	version number = B1 symbol index = B2 to B5 (symbol-relative loc to patch w/ line table offset) offset = B6 to B9 (symbol + offset, location to patch w/line table offset)
R_LINETAB_ESC	0xDB	3	escape code = B1 number of following R_STATEMENT fix-ups containing escape data in B2
R_LTP_OVERRIDE	0xDC	1	none
R_COMMENT	0xDD	6	OP=B1 V = B2 to B6
R_TP_OVERRIDE	0xDE	1	none
R_RESERVED	0xDF-FF	-	reserved

3.6.7 Parameter Relocation Bits (rbits1, rbits2)

Parameter relocation bits are encoded in the fixup requests in two ways, noted as rbits1 and rbits2 in Table 15. The first encoding recognizes that the most common procedure calls have only general register arguments with no holes in the parameter list. The encoding for such calls is simply the number of parameters in general registers (0 to 4), plus 5 if there is a return value in a general register.

Here is how “rbits1” decodes its parameter. The “diff” is the difference between the actual opcode and the first opcode of the range. When “rbits1” is used, it is describing a function call with 0 to 4 general register parameters (no holes, and no floating point register parameters), and either a general register return value or no return value. The “diff” can be from 0 to 9; if it is between 0 and 4, it indicates 0 to 4 parameters with no return value; if it’s between 5 and 9, it indicates 0 to 4 parameters with a return value. Here is some code that turns this into the 10-bit parameter relocation field:

```

if (diff >= 5)

j = diff -5;

else

j = diff;

for (i = 0; i < 4; i++)

```

```
arg = (arg << 2) + (i<j);
```

```
arg = (arg << 2) + (diff >=5);
```

The second encoding is more complex (presumably less common); the 10 argument relocation bits are compressed into 9 bits by eliminating some impossible combinations. The encoding is the combination of three contributions. The first contribution is the pair of bits for the return value, which are not modified. The second contribution is 9 if the first two parameter words together form a double-precision parameter; otherwise, it is 3 times the pair of bits for the first word plus the pair of bits for the second word. Similarly, the third contribution is formed based on the third and fourth parameter words. The second contribution is multiplied by 40, the third is multiplied by 4, then the three are added together. Here is some code to decode the “rbits” encoding:

```
i = ((diff &1) <<8) + next_fixup_byte;
```

```
arg = decode_arg_reloc(i);
```

where:

```
int decode_arg_reloc(i)
```

```
int i;
```

```
{
```

```
int j, k, ret_val;
```

```
ret_val = i &03;
```

```
i >>= 2;
```

```
j = i / 10;
```

```
i -= 10*j;
```

```
if (j == 9)
```

```
ret_val += (03 << 6);          /* FARGU */
```

```
else {
```

```
k = j / 3;
```

```
j -= 3*k;

ret_val += (k << 8) + (j << 6);

}

if (i == 9)

ret_val += (03 << 2);           /* FARGU */

else {

k = i / 3;

i -= 3*k;

ret_val += (k << 4) + (i << 2);

}

return (ret_val);

}
```

3.7 Symbol Table

The symbol table or symbol dictionary for a SOM consists of symbol records strung together in contiguous space within the SOM. The byte offset of the dictionary, relative to the SOM header, is contained in the variable *symbol_dictionary_location* in the SOM header and the number of entries is contained in the variable *symbol_dictionary_total*, also in the SOM header.

A particular symbol in the dictionary can be located either by scanning the dictionary until it is found, or the symbol's index can be used to index into the dictionary as if it were an array of five word elements.

Note

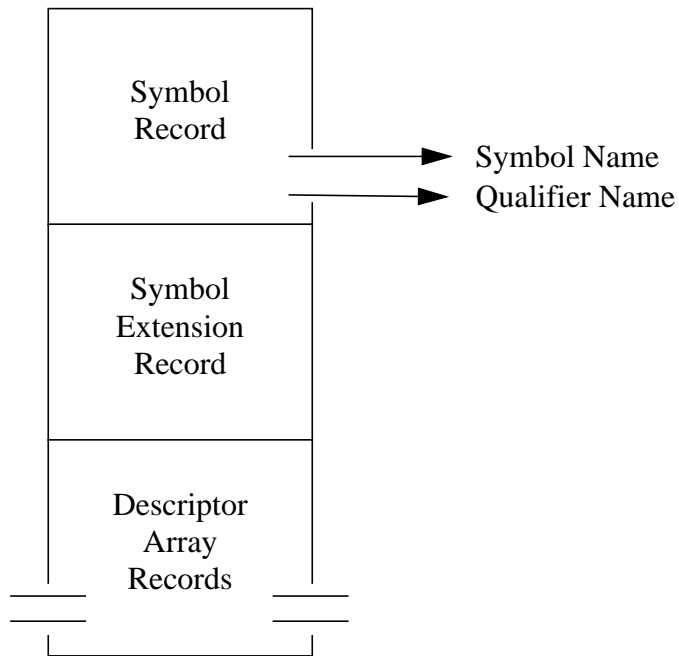

A symbol's index is NOT its relative entry number in the symbol dictionary since some entries use extension records and argument descriptor arrays. But all entries are a multiple of 5-words in length so the index can be used to index into the symbol dictionary.

Figure 2-13: Symbol Dictionary Record Definition

```
struct symbol_dictionary_record {
    unsigned int    hidden                : 1;
    unsigned int    secondary_def         : 1;
    unsigned int    symbol_type           : 6;
    unsigned int    symbol_scope          : 4;
    unsigned int    check_level           : 3;
    unsigned int    must_qualify          : 1;
    unsigned int    initially_frozen      : 1;
    unsigned int    memory_resident       : 1;
    unsigned int    is_common             : 1;
    unsigned int    dup_common            : 1;
    unsigned int    xleast                : 2;
    unsigned int    arg_reloc             :10;
    union name_pt   name;
    union name_pt   qualifier_name;
    unsigned int    has_long_return       :1;
    unsigned int    no_relocation         :1;
    unsigned int    is_comdat             :1;
    unsigned int    reserved              :5;
    unsigned int    symbol_info            :24;
    unsigned int    symbol_value;
};
```

An entry in the dictionary consists of the symbol dictionary record and an optional extension record and 0 to 61 descriptor array records as shown in Figure 2-13. Symbol records do not need to be sorted.

Figure 2-14: Structure of a Dictionary Entry



Whether an extension record and argument descriptor arrays follow the symbol record is dependent upon the check level and the number of parameters according to the following algorithm:

```
IF CHECK_LEVEL >= 1
  THEN
```

```
    An extension record will be present.
```

```
    IF num_args > 3 AND check_level >= 3
      THEN
```

```
        In addition to the extension record there will be enough
        argument descriptor arrays to contain one descriptor for each argument except the first 3.
        i.e. NUM_DESCS = round_up ( (NUM_ARGS-3)/4 )
```

Two symbol types, SYM_EXT and ARG_EXT, are defined to mark the symbol extension and argument descriptor array records respectively.

hidden

Bit 0

If this flag is set to one, it indicates that the symbol is to be hidden from the loader for the purpose of resolving external (inter-SOM) references. It has no effect on linking. This flag allows a procedure to be made private to its own executable SOM, although it has universal scope within that SOM.

secondary_def

Bit 1

If this flag is set to one, the symbol is a secondary definition and has an additional name that is preceded by “_”. The linker will ignore duplicate definitions involving secondary definitions.

symbol_type

Bits 2-7

This field defines what type of information this symbol represents. A complete list of the defined symbol types is presented in Table 16, however only certain ones may be valid depending on the use (e.g. import/export, relocatable/executable, etc.).

TABLE 16

symbol_type Definition

#	Symbol	Description
0	NULL	Invalid symbol record. The contents of the entire record is undefined (it is 5 words long).
1	ABSOLUTE	Absolute constant.
2	DATA	Normal initialized data. Initialized data symbols including Fortran and Cobol initialized common data blocks, as well as C initialized data. Data can be either imported or exported. For example C construct “EXTERN INT I” would be imported data. And the C construct “INT I = 1” would be exported data.
3	CODE	Unspecified code. For example, code labels. Code labels are only relevant up to link time, and they cannot be the target of interspace calls.
4	PRI_PROG	Primary program entry point.
5	SEC_PROG	Secondary Program entry point.
6	ENTRY	Any code entry point. Includes both primary and secondary entry points. Code entry point symbols may be used as targets of inter-space calls.
7	STORAGE	The value of the symbol is not known, but the length of the area is given. If a matching definition is not found, storage is allocated within a specified subspace and the symbol’s value becomes the virtual address of that storage. For example, Fortran and Cobol uninitialized common data blocks, and the C construct “INT I” would be storage requests with no initial value.

TABLE 16

symbol_type Definition

#	Symbol	Description
8	STUB	This symbol marks an import (outbound) external call stub (EXTERNAL scope) or a parameter relocation stub (LOCAL scope). The linker may create an import stub for any unsatisfied code symbols, and the loader would be responsible for satisfying the reference by filling in the XRT entry allocated for this stub.
9	MODULE	This symbol is a source module name.
10	SYM_EXT	This type is used to indicate that an entry in the SOM symbol dictionary is an extension record of the current entry (previous valid symbol entry in the list).
11	ARG_EXT	This type is used to indicate that an entry in the SOM symbol dictionary is an extension record of the current entry (previous valid symbol entry in the list).
12	MILLICODE	This is the name of the millicode routine.
13	PLABEL	This symbol defines an export stub for a procedure for which a procedure label has been generated. The loader must build an XRT entry for the procedure at the offset allocated by the linker.
14	OCT_DIS	This type is used to indicate that the pointer to a translated code segment exists, but has been disabled. Used by the Object Code Translator only.
15	MILLI_EXT	This symbol defines the address of an external millicode subroutine. It should be treated as an constant.
16	TSTORAGE	This symbol defines Thread Specific data storage.
17	COMDAT	This type is used to identify the secondary subspaces of a COMDAT set to support the C++ Compile Time Template Instantiation feature

symbol_scope

Bits 8-11

The scope of a symbol defines the range over which an exported symbol is valid, or the range of the binding used to import the symbol. In addition, this field is used to determine whether the requested symbol record is a import or export request.

The scope of a symbol will be one of the following:

Imports

0: UNSAT Import request that has not been satisfied.

1: EXTERNAL Import request linked to a symbol in another SOM. This symbol will require additional linking when it is loaded.

Internal

2: LOCAL This symbol is not exported for use outside the SOM. It may be used as the target for fixups, but the linker does not use this symbol for resolving symbol references.

Exports

3: UNIVERSAL This symbol is exported for use outside the SOM.

Table 17 shows the valid values of the scope field given the type of the symbol. Any square that does not contain an “X” is an invalid value for that type.

TABLE 17

Valid *symbol_scope* Values

TYPE	UNSAT	EXTERNAL	LOCAL	UNIV
PRI_PROG				X
SEC_PROG				X
ENTRY			X	X
STUB		X	X	
MODULE			X	X
ABSOLUTE	X		X	X
CODE	X		X	X
DATA	X		X	X
STORAGE	X			
PLABEL			X	

check_level

Bits 12-14

This value indicates how closely an import definition must match an export definition during linking. This checking can be applied to both code and data linkage according to the following checking levels:

- 0 No checking.
- 1 Check the symbol type descriptor only.
- 2 Level 1, plus check the number of arguments passed by the import with the minimum and maximum range declared in the export (code types only).
- 3 Level 2, plus check the type of each argument passed (code types only).

must_qualify

Bit 15

If this bit is set to one, it indicates that there is more than one entry in the symbol directory that has the same name as this entry, and is the same generic type (i.e. code, data or stub). Therefore, the qualifier name must be used to fully qualify the symbol.

If this flag is not set, the qualifier name will only be used to qualify the symbol name if the name it is being compared with is also fully qualified.

The flag is used for both import and export requests.

initially_frozen

Bit 16

If this flag is set to one it indicates that the code importing or exporting this symbol is to be locked in physical memory when the operating system is being booted.

memory_resident

Bit 17

If this field is set to one it indicates that the code that is importing or exporting this symbol is frozen in memory. This flag is used so that links between memory resident procedures can also be frozen in memory.

is_common

Bit 18

Specifies that this symbol is an initialized common data block. Each initialized common data block resides in its own subspace. For example, a Fortran initialized common declaration would produce a symbol of type data with the *is_common* flag set to one. Refer to the Language Requirements Document for implementation details.

dup_common

Bit 19

If this flag is set to one, it specifies that this symbol name may conflict with another symbol of the same name if both are of type data. This is to facilitate the Cobol “common” feature, since Cobol allows duplicate initialization of “common” data blocks. This flag would be set to one if the language allows duplicate initialization, otherwise it will be set to zero for symbols of type data. Refer to the Language Requirements Document for implementation details.

xleast

Bits 20-21

This is the execution level that is required to call this entry point. This XLEAST level is placed in any XRT entry linked to this entry point. The XLEAST level will be checked by the Spectrum external procedure call primitive during execution.

This field is not used if (i.e. its content is meaningless):

- 1) the symbol is an import.
- 2) the symbol is not one of the code types.

XLEAST must be a value in the range of 0 to 3. Furthermore, if the value is not in the range of XLEAST to XMOST of the page containing the entry point a run time error can occur.

arg_reloc

Bits 22-31

This field is used to 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. This field is meaningful only for exported ENTRY, PRI_PROG, and SEC_PROG symbols.

The linker matches the argument relocation bits of an exported symbol with the argument relocation bits in each fixup that references the symbol. If it finds a mismatch, it builds an argument relocation stub and redirects the call to that stub.

The ten bits of this field are broken down as follows:

- 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 return

For MPE/iX, this field can contain new values if the shared_data bit in the LST SOM Auxiliary header is set:

For Storage requests and Data Universals, this field is set to the access rights of the subspace the data is defined in.

For Data Unsats, this field is set to the access rights of the subspace that contains the reference.

The argument location value is defined as follows:

Table 5-1:

Value	Mnemonic	Location
0		Do not relocate - Mismatch is not an error.
1	ARG	Argument Register
2	FARG	Floating point coprocessor register, bits 0 to 31.
3	FARGupper	Floating point coprocessor register, bits 32 to 64.

The FARGupper tag can be used only for parameter list words 0 and 2, or for the function return. If it is used for parameter list words 0 or 2, then parameter list word 1 or 3, respectively, must be tagged as FARG; this indicates a double-precision floating-point number in a single floating point coprocessor register. If it is used for the function return, it indicates a double-precision floating point return value in a single floating point coprocessor register.

name

This variable is used to locate the name of the symbol in the symbol dictionary string table of the SOM. Its value is the byte offset, relative to the beginning of the string table, to the first character (not the length) of the symbol name. The name begins on a word boundary and is preceded by a 32 bit number that contains the number of characters in the name. The symbol is terminated with an 8 bit zero, but the terminator is not included as part of the length.

The size of the symbol dictionary string area can be used to bounds check this variable such that it is a value in the range of 0 to the value of the variable *symbol_strings_size* found in the SOM header.

qualifier_name

This field contains a byte offset relative to the beginning of the symbol strings area which points to the first character of a symbol name which may be used to further qualify the current symbol.

If there is no qualifier, this field should be set to 0.

has_long_return

this bit is set for an Entry Universal symbol means its return sequence can cross a space; setting it for a Code Unsat asserts that the called entry point will have a long return sequence.

no_relocation

Setting the no_relocation bit for an Entry Universal is unnecessary, but should be done for any such symbol record whose arg_reloc field is 0. Setting the bit for a Code Unsats asserts that the called entry point will not require any parameter relocation.

is_comdat

Setting the is_comdat bit to identify the symbol as the key symbol for a set of COMDAT subspaces

symbol_info

This field contains variant information depending on the scope of the symbol. The following list shows the interpretation of this field:

Table 5-2:

Scope	Meaning
UNSAT	Contains the index of the subspace that imported this symbol. If STORAGE_REQUEST then it is the index of the subspace which may contain this symbol.
EXTERNAL	Contains the XRT offset allocated by the linker for the import stub.
LOCAL	Index of the subspace containing this symbol. For export stubs (procedure labels), this field contains the XRT offset instead.
UNIVERSAL	Index of the subspace containing this symbol.

For MPE/iX, this field can contain new values if the shared_data bit in the LST SOM Auxiliary header is set:

For Storage Requests, this field is set to the size of the storage request.

For Data Unsats, this field is set to the index of the subspace that contained the reference.

symbol_value

This field contains the 32 bit value of this particular symbol.

Depending on the type and scope of the symbol this field may have a different meaning. The following matrix shows the meaning of the symbol value for each valid combina-

tion of type and scope. Invalid combinations will be denoted as a blank cell in the matrix. Immediately following the matrix are the definitions for the mnemonics used.

TABLE 6
Valid symbol_value Mnemonics

TYPE	UNSAT	EXTRN	LOCAL	UNIV
PRI_PROG				SOFF
SEC_PROG				SOFF
ENTRY			SOFF	SOFF
STUB		SOFF	SOFF	
MODULE			UNUSED	UNUSED
ABSOLUTE	UNUSED		CONST	CONST
CODE	UNUSED		SOFF	SOFF
DATA	UNUSED		SOFF	SOFF
STORAGE	LEN			
PLABEL			SOFF	

SOFF - This stands for space offset and it is the byte offset within a space (when it is loaded in virtual memory) to an entry point (i.e. the first instruction to be executed). For code symbols, bits 30-31 of the offset will contain the privilege level that the procedure will execute at (subject to privilege level checking at load time and XLEAST / XMOST level checking during execution).

CONST - This stands for a numeric constant or its value may be the virtual address of a location within a subspace defined by this SOM.

LEN - This is the length of the storage request in bytes.

UNUSED- The content of this field is meaningless.

For MPE/iX, this field can contain new values if the shared_data bit in the LST SOM Auxiliary header is set:

For Data and Storage Universals, this field is set to the DP-positive address of the symbol.

For Data Unsats, this field is set to the DXRT offset for the symbol (will be a negative offset).

Symbol Dictionary Extension Record

Figure 2-15:

```
struct symbol_extension_record {
    unsigned int    type                :8;
    unsigned int    max_num_args       :8;
    unsigned int    min_num_args       :8;
    unsigned int    num_args           :8;
    union arg_descriptorsymbol_desc;
    union arg_descriptorargument_desc[3];
};
```

Symbol Dictionary Extension Record Fields

type

Bits 0-7

This field will be set to SYM_EXT (i.e. 12) so that it can be identified as an extension to the symbol definition of the previous entry in the symbol list (see SYMBOL_TYPE of symbol dictionary record fields).

max_num_args

Bits 8-15

If CHECK_LEVEL indicates that the number of arguments passed should be checked, the num_args field of the imported symbol (this field is in the exported symbol) must be less than or equal to this value.

This field is not used if (i.e. its content is meaningless) if the symbol is an import.

The range of this variable is min_num_args to 255.

min_num_args

Bits 16-23

If CHECK_LEVEL indicates that the number of arguments passed should be checked, the num_args field of the imported symbol (this field is in the exported symbol) must be greater than or equal to this value.

This field is not used if (i.e. its content is meaningless) if the symbol is an import.

The range of this variable is 0 to max_num_args".

num_args

Bits 24-31

This value is the number of arguments associated with the symbol. A procedure return value is NOT counted as an argument.

The range of this variable is 0 to 255. Since this variable is not essential for linking or loading, compilers are not constrained to limit the number of parameters to 255. However, if this limit is exceeded, functions that use this field (e.g. parameter checking) may produce unpredictable results.

symbol_desc

This is an argument descriptor for the procedure's type or the data type depending upon the type of the symbol (see argument descriptor definition, section 9.5).

This field is not used (i.e. its content is meaningless) if the checking level is 0.

argument_desc [1]

This is the argument descriptor for the first argument in the procedure's argument list.

This field is not used (i.e. its content is meaningless) if the checking level is less than 3 or the number of arguments is 0.

argument_desc [2]

This is the argument descriptor for the second argument in the procedure's argument list.

This field is not used (i.e. its content is meaningless) if the checking level is less than 3 or the number of arguments is less than 2.

argument_desc [3]

This is the argument descriptor for the third argument in the procedures argument list.

This field is not used (i.e. its content is meaningless) if the checking level is less than 3 or the number of arguments is less than 3.

Argument Descriptor

```

struct argument_desc_array {
    unsigned int          type : 8;
    unsigned int          reserved : 24;
    union arg_descriptor  argument_desc[4];
};

```

Argument Descriptor Fields

Figure 2-16: Argument Descriptor Definition

```

union arg_descriptor {
    struct {
        unsigned int    reserved      :3;
        unsigned int    packing       :1;
        unsigned int    alignment    :4;
        unsigned int    mode          :4;
        unsigned int    structure     :4;
        unsigned int    hash          :1;
        int             arg_type      :15;
    } arg_desc;
    unsigned int        word;
};

```

reserved

Bits 0-2

These bits are reserved for future use, and must be set to zero.

packing

Bit 3

This field specifies the packing algorithm used in calculating the storage layout, the alignment of, and the data representation of the particular item. The real number data representation on Spectrum is different from that of the HP 3000. This field may be increased in size to allow more packing possibilities, such as 9000 or 1000 packing. The valid values for this field are:

- | | |
|---|--|
| 0 | Spectrum packing and IEEE real numbers |
| 1 | 3000 mode packing and real numbers alignment |

Bits 4-7

This field specifies the alignment of the descriptor. The valid values for this field are:

0	Byte aligned
1	Half-word aligned
2	Word aligned
3	Double-word aligned
4	Cache line (2^4 byte, 16-byte) aligned
5	Cache line (2^5 byte, 32-byte) aligned
..	..
n	Cache line (2^n byte) aligned
..	..
12	Page (2^{12} byte, 4096-byte) aligned

mode

Bits 8-11

This field specifies the type of the descriptor and its use. A value of zero for this field is used to match with any other value. The valid values for this field are:

0	Wild card
1	Parameter, passed by value
2	Parameter, passed by reference
3	Parameter, passed by value-result
4	Parameter, passed by name
5	Global/External/Module variable
6	Function return
7	Procedure
8	Parameter, passed by long reference

structure

Bits 12-15

This field specifies the structure for a particular item.

A value of zero for this field will match any other value.

The valid values for this field are:

0	Wild card
1	Simple variable
2	Array
3	Record or composite

4	Short pointer
5	Long pointer
6	String, zero terminated
7	String, with length word
8	Procedure
9	Function
10	Label

hash

Bit 16

This bit, when set, specifies that the `arg_type` field contains a hash value, rather than a predefined type.

arg_type

Bits 17-31

This field specifies the basic machine type for the particular item. If the item is a record, string, or procedure (structure field 3, 6, 7, or 8), the type will be void. Type 17 (structure or array) is allowed only when the structure field is type 2 (array), which describes an array or structure within an array. A value of zero for this field is used to match with any other valid value. The valid values for this field are:

0	Wild card
1	Void
2	Signed byte(8 bits)
3	Unsigned byte(8 bits)
4	Signed half-word(16 bits)
5	Unsigned half-word(16 bits)
6	Signed word(32 bits)
7	Unsigned word(32 bits)
8	Signed double-word(64 bits)
9	Unsigned double-word(64 bits)
10	Short real(32 bits)
11	Real(64 bits)
12	Long real(128 bits)
13	Short complex(64 bits)
14	Complex(128 bits)
15	Long complex(256 bits)
16	Packed decimal
17	Structure or array

A relocatable library is a file of one or more SOMs and the data structures needed to efficiently manage the SOMs. At the front of the file is a Library Symbol Table (LST) header. The header is used to identify the file structure and locate the major sub-structures of the library. In particular, the header contains the location of the symbol directory, the SOM directory, an optional area for auxiliary headers and the free space list.

Figure 2-17 on page 98 shows a general layout of a relocatable library. Note that each

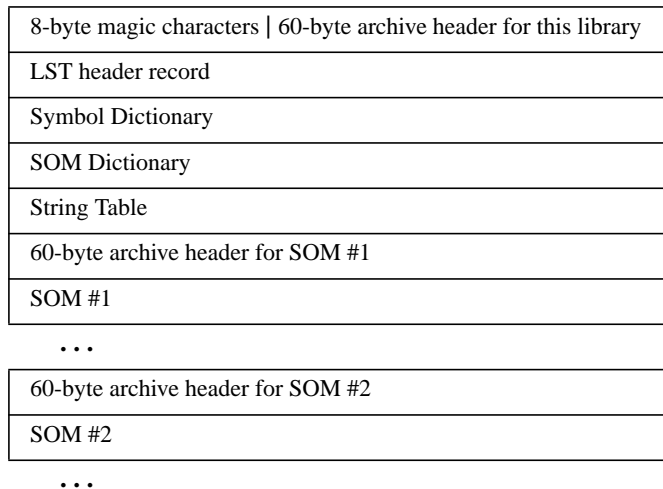


Figure 2-17: General Layout of a Relocatable Library

relocatable library is identifiable by the following 8-byte magic characters at the beginning of the file:

!<arch>\n

where\n is the new line, or the line feed character (hex 0A).

4.1 Archive Header

The archive header appears in front of every SOM in a library, and in front of the LST in a relocatable library. It defines the name of the SOM that follows and its length (in bytes), as well as several other fields that are used by the HP-UX archiver utility. See the HP-UX Users' Manual for further details.

ar_name

This field contains the name of the following SOM. The name is that of the “.o” file that was copied into the library. The name must be left justified in the field, terminated by a slash (“/”), and padded on the right with blanks.

For the archive header that precedes the LST, this field should contain a blank name (i.e., a single slash padded with 15 blanks).

If a member with a file name greater than 15 bytes exists within the archive, then the archive will also contain an additional special member to store the long file name string table. The special string table member also has a zero length name where `ar_name[0] == '/'` and `ar_name[1] == '/'`.

```

struct ar_hdr { /* archive file member header - printable ascii */
    char          ar_name[16];          /* file member name - '/'
terminated */
    char          ar_date[12];          /* file member date - decimal */
    char          ar_uid[6];            /* file member user id - decimal
*/
    char          ar_gid[6];            /* file member group id -
decimal */
    char          ar_mode[8];           /* file member mode - octal */
    char          ar_size[10];          /* file member size - decimal */
    char          ar_fmag[2];           /* ARFMAG - string to end
header */
};

```

Figure 2-18: Definition of Archive Header Record

If a special string table exists, it will precede all non-special archive members. If both a symbol table member and a string table member exist then the symbol table member will always precede the string table member.

Each entry in the string table is followed by a slash and a new-line character. The offset of the table begins at zero. If an archive member name exceeds 15 bytes, then the `ar_name` entry in the members header does not hold a name, but holds the offset into the string table preceded by a slash.

For example, the member name *thisverylongfilename.o* contains /0 for the `ar_name` value. This value represents the offset into the string table. The member name *yetanotherfilename.o* contains /27 for the `ar_name` value. The long name string table would have the following format:

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	t	h	i	s	i	s	a	v	e	r
10	y	l	o	n	g	f	i	l	e	n
20	a	m	e	.	o	/	\n	y	e	t
30	a	n	o	t	h	e	r	l	o	n
40	g	f	i	l	e	n	a	m	e	.
50	o	/	\n							

ar_date

This field contains the modification date and time of the following SOM or LST. It should be a decimal number (in ASCII characters) representing the number of seconds since January 1, 1970. The number should be left adjusted in the field and padded with blanks.

ar_uid

This field contains the user id of the owner of the following SOM or LST. It should be a decimal number (in ASCII), left adjusted and blank padded.

ar_gid

This field contains the group id of the owner of the following SOM or LST. It should be a decimal number (in ASCII), left adjusted and blank padded.

ar_mode

This field contains the mode bits for the following SOM or LST. It is an octal number, left adjusted and blank padded.

ar_size

This field contains the size of the following SOM or LST in bytes. It is an ASCII decimal number, left adjusted and blank padded. The size does not include the archive header.

ar_fmags

This field always contains the two ASCII characters “” and newline (or line feed, hex 0A).

4.2 Library Symbol Table Header Record

The Library Symbol Table always begins with a LST header record. For a relocatable library, the LST header begins immediately following the 8-byte archive “magic string” and the 60-byte archive header; the file name field in the archive header is empty (i.e., “/” followed by 15 blanks).

The first four bytes of the LST header will contain a number that identifies the file as a library format file (actually it has a sub-structure of two 16 bit numbers). In addition, the header is used to locate the major sub-structures of the library. In particular, the header contains the locations of the symbol directory, the SOM directory, the import table which is always set to zero, an optional area for auxiliary headers and the free space list.

system_id

```

struct lst_header {
    short int    system_id;
    short int    a_magic;
    unsigned int version_id;
    struct sys_clock file_time;
    unsigned int hash_loc;
    unsigned int hash_size;
    unsigned int module_count;
    unsigned int module_limit;
    unsigned int dir_loc;
    unsigned int export_loc;
    unsigned int export_count;
    unsigned int 0 (import_loc);
    unsigned int aux_loc;
    unsigned int aux_size;
    unsigned int string_loc;
    unsigned int string_size;
    unsigned int free_list;
    unsigned int file_end;
    unsigned int checksum;
};

```

Figure 2-19: LST Header Definition

Bits 0-15

This field is used to identify the architecture that this object file is targeted for. The PA-RISC 1.1 architecture *system_id* is 210 (hexadecimal).

a_magic

Bits 16-31

This is a number that indicates the format and function of the file.

The magic number for a relocatable library is 0619 (hex), and for an executable library is 0104 (hex).

version_id

This is a number that is used to associate the LST with the correct definition of its internal organization. The value of the number will be an encoding of the date the LST version was defined.

The version ID can be interpreted by viewing it in decimal form and separating it into character packets of YYMMDDHH, where YY is the year, MM is the month, DD is the day, and HH is the hour.

The only version_id that is currently defined for use by conforming applications is 85082112.

file_time

file_time is a 64 bit value that represents the time the file was last modified. *file_time* is actually composed of two 32 bit quantities where the first 32 bits is the number of seconds that have elapsed since January 1, 1970 (at 0:00 GMT), and the second 32 bits is the nano second of the second (which requires 30 bits to represent).

This value is independent of any modification time maintained by other subsystems (e.g. the file system). The use of this field is optional, but if it is not used it will be set to zero.

hash_loc

This is the LST relative byte offset to the LST directory hash table.

hash_size

This is the number of entries in the LST directory hash table.

Since the number of entries in the hash table is also the number of symbol lists in the directory, changing this value can affect the length of the symbol lists. The length of the symbol lists in turn, affects the overhead required to locate a symbol.

This value must be a number between 1 and $2^{31}-1$. The maximum size of the hash table is not constrained by the range of this variable, but by other resource constraints (e.g. file size).

module_count

This contains the index beyond the last used SOM directory entry.

module_limit

This is the maximum number of SOMs that can be in this file. Therefore, it is also the number of entries in the SOM directory table and the number of entries in the import table.

This value must be a number between 1 and $2^{31}-1$. The maximum value of this variable will be constrained by external resource constraints (e.g. system tables with SOM reference counts may use fixed length arrays).

dir_loc

This is the LST relative byte offset to the SOM directory.

export_loc

This is the LST relative byte offset to the export table. Not all exported symbols are necessarily contained within the bounds defined by *export_loc* and *export_count*, but most symbols should be. These fields are provided to allow programs that process the export table to read in the majority of the symbol table efficiently.

export_count

This is the number of symbols contained in the main portion of the export table. Overflow symbols (symbols allocated after this table is full) may be scattered throughout the LST.

import_loc

This is the LST relative byte offset to the import table. It is set to zero for relocatable library.

aux_loc

This is the LST relative byte offset to the auxiliary header area. If no auxiliary headers are present this variable will be set to zero.

aux_size

This is the size of the auxiliary header area in bytes. If no auxiliary headers are present this variable will be set to zero.

string_loc

This is the LST relative byte offset to the string area of the LST.

string_size

This is the size of the LST string area in bytes.

free_list

This is the LST relative byte offset to the first free area in the file.

file_end

This is the LST relative offset to the first byte past the end of the file.

checksum

This field contains the value of all the other fields (i.e. not including this field) in the LST header record after they have been exclusive ORed together.

If (in the future) there is are undefined bits in this record they must be set to zero so that they do not affect the value of *checksum*.

4.3 Library Symbol Table Format

Data structures in relocatable library are designed to efficiently manage the SOMs in the library. The LST header record contains addresses and ranges of the sub-structures inside the library. Symbol dictionary and SOM dictionary are the two most important data structures of a relocatable library. describes a relocatable as a block diagram seen from the LST header record, the rest of this section describes the data structures in the relocatable library.

4.3.1 Symbol Directory

The symbol directory provides direct access to the definitions of all the exported symbols in the library. Each symbol definition, in turn, contains the index number of the SOM that exported the symbol. The SOM index can be used to index into the SOM directory or the import table (to locate the SOM or its import list).

The LST directory search algorithm will support more than one entry with the same name provided it can be qualified by its module name or by the general type of the symbol (i.e. code, data or stub).

The symbol directory is implemented as a hash table. Each entry contains an offset to a “hash bucket” which is a chained list of symbols that hash to the same index. If a bucket

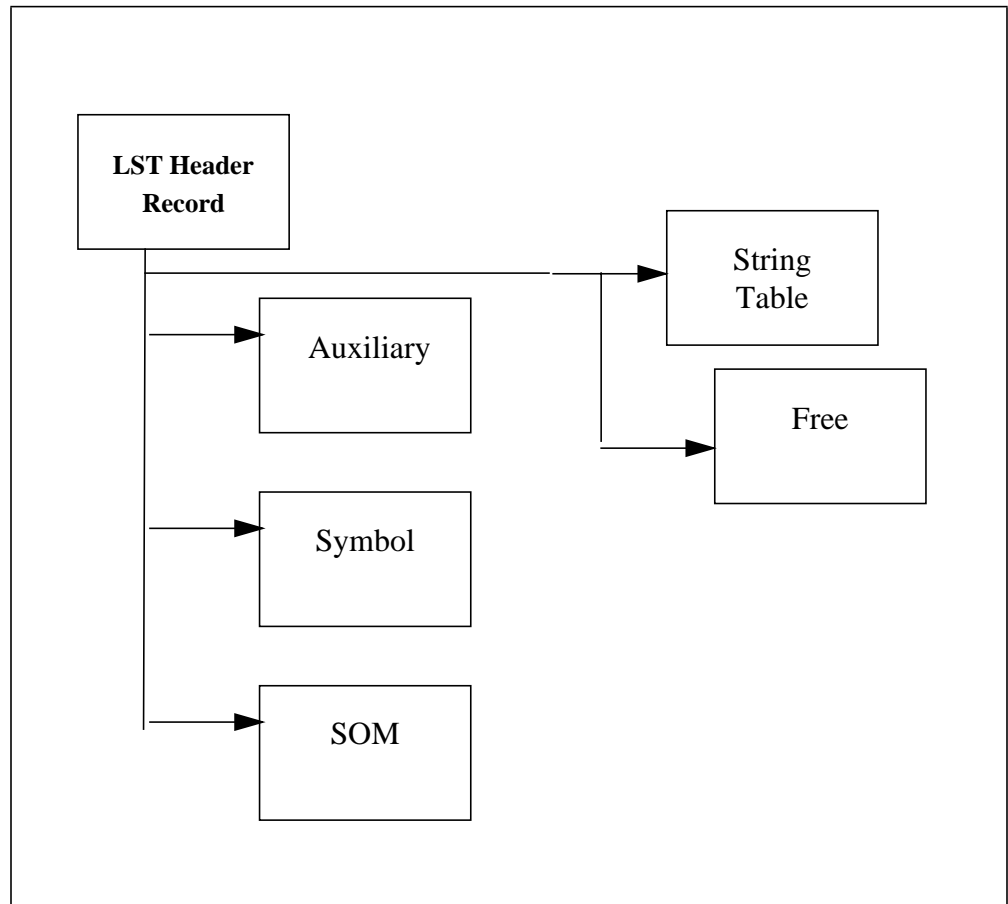


Figure 2-20: Block Diagram of a Relocatable Library

is empty, its hash table entry will be zero and the bucket will not exist. The number of entries in the hash table is contained in the variable *hash_size* in the LST header and the hash table location is contained in the variable *hash_loc*.

The hash function that is used for indexing the symbol directory is *hash_key* modulo *hash_size*. The hash key is a 4 byte variable where the first byte is the length of the symbol, the second byte of the key is the second character in the symbol, the third byte of the key is the next to last character in the symbol, and the last byte of the key is the last character in the symbol. If the symbol is only one character long, then that character is used as the second byte of the key and the last two bytes of the key are the same as the first two bytes. The result of the hash function is the hash table entry number, not the offset into the hash table.

Note

If a symbol is greater than 128 characters the first byte of the key will be the symbol length modulo 128 (256 is not used to eliminate any affect the sign bit may have on the modulo operation).

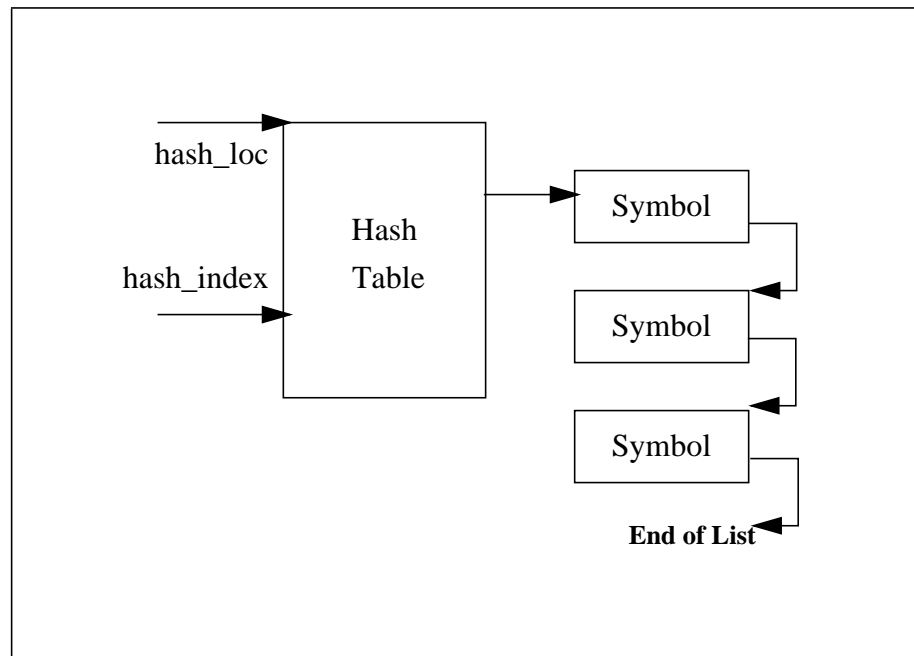


Figure 2-21: Block Diagram of Symbol Directory

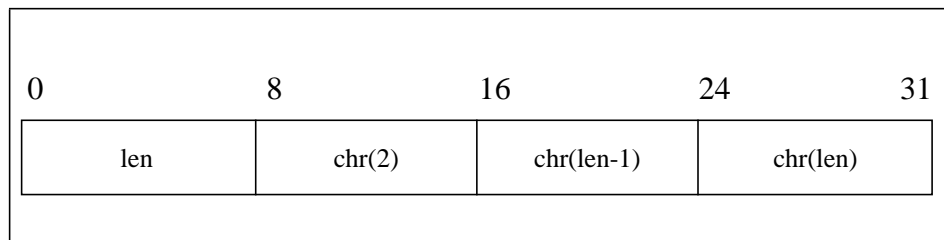


Figure 2-22: hash_key Format (symbol length > 1 byte)

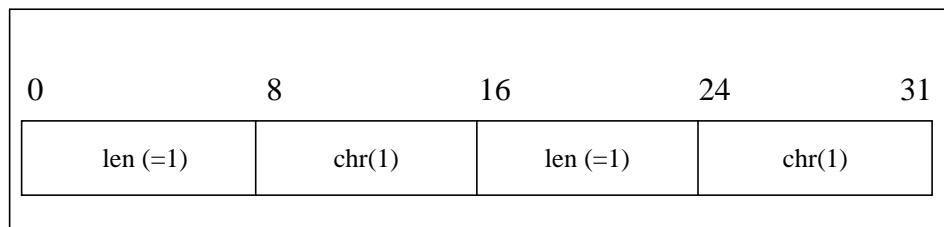


Figure 2-23: hash_key Format (symbol length = 1 byte)

A symbol record consists of a symbol header record and 0 to 255 argument descriptors constructed as shown in Figure 2-24.

Symbol records are used for the symbol entries in both the LST symbol directory and the import list symbol entries.

The symbol header contains the information needed to import or export a symbol when the file is loaded. The presence and number of argument descriptors is determined by a combination of the checking level and the number of arguments according to the following algorithm:

```
IF CHECKING_LEVEL < 3
THEN
    No argument descriptors present (except the symbol descriptor
    in the header).
ELSE
    There will be one descriptor for every argument passed.
```

The `lst_symbol_record` structure

In general, the `lst_symbol_record` structure is very similar to the `symbol_dictionary_record` of the SOM with the addition of the `som_index`, `symbol_key`, and `next_entry` fields to support symbol searching.

hidden

Bit 0

If this flag is set to one, it indicates that the symbol is to be hidden from the loader for the purpose of resolving external (inter-SOM) references. It has no effect on linking. This flag allows a procedure to be made private to its own executable SOM, although it has universal scope within that SOM.

secondary_def

Bit 1

If this flag is set to one, the symbol is a secondary definition and has an additional name that is preceded by “_”. The linker will ignore duplicate definitions involving secondary definitions. This field is implemented to support the external naming convention.

```
struct lst_symbol_record {
    unsigned int    hidden                : 1;
    unsigned int    secondary_def         : 1;
    unsigned int    symbol_type           : 6;
    unsigned int    symbol_scope          : 4;
    unsigned int    check_level           : 3;
    unsigned int    must_qualify          : 1;
    unsigned int    initially_frozen      : 1;
    unsigned int    memory_resident       : 1;
    unsigned int    is_common             : 1;
    unsigned int    dup_common            : 1;
    unsigned int    xleast                : 2;
    unsigned int    arg_reloc             :10;
    union name_pt   name;
    union name_pt   qualifier_name;
    unsigned int    symbol_info;
    unsigned int    symbol_value;
    unsigned int    symbol_descriptor;
    unsigned int    reserved              : 8;
    unsigned int    max_num_args          : 8;
    unsigned int    min_num_args          : 8;
    unsigned int    num_args              : 8;
    unsigned int    som_index;
    unsigned int    symbol_key;
    unsigned int    next_entry;
};
```

Figure 2-24: LST Symbol Record Definition

symbol_type

Bits 2-7

This field defines what type of information this symbol represents.

See “Symbol Table” on page 80.

symbol_scope

Bits 8-11

The scope of a symbol defines the range over which an exported symbol is valid, or the range of the binding used to import the symbol. In addition, this field is used to determine whether the symbol record is a import or export request.

See “Symbol Table” on page 80.

check_level

Bits 12-14

This value indicates how closely an import definition must match an export definition during linking.

For more info on `check_level`, See “Symbol Table” on page 80..

must_qualify

Bit 15

If this bit is set to one, it indicates that there is more than one entry in the symbol directory that has the same name as this entry, and is the same generic type (i.e. code, data, or stub). Therefore, the qualifier name must be used to fully qualify the symbol.

If this flag is not set, the qualifier name will only be used to qualify the symbol name if the name it is being compared with is also fully qualified.

must_qualify is used for both import and export requests.

initially_frozen

Bit 16

If this flag is set to one it indicates that the code importing or exporting this symbol is to be locked in physical memory when the operating system is being booted.

memory_resident

Bit 17

If this field is set to one it indicates that the code that is importing or exporting this symbol is frozen in memory. This flag is used so that links between memory resident procedures can also be frozen in memory.

is_common

Bit 18

Specifies that this symbol is an initialized common data block. Each initialized common data block resides in its own subspace. For example, a Fortran initialized common declaration would produce a symbol of type data with the *is_common* flag set to one.

duplicate_common

Bit 19

If this flag is set to one, it specifies that this symbol name may conflict with another symbol of the same name if both are of type data. This is to facilitate the Cobol “common” feature, since Cobol allows duplicate initialization of “common” data blocks. This flag would be set to one if the language allows duplicate initialization, otherwise it will be set to zero for symbols of type data.

xleast

Bits 20-21

This is the execution level that is required to call this entry point. This *xleast* level is placed in any XRT entry linked to this entry point. The *xleast* level will be checked by the Spectrum external procedure call primitive during execution.

See “Symbol Table” on page 80.

arg_reloc

Bits 22-31

This field is used to 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. This field is meaningful only for exported ENTRY, PRI_PROG, and SEC_PROG symbols.

See “Symbol Table” on page 80.

name

This variable is used to locate the name of the symbol in the string table of the LST. Its value is the byte offset, relative to the beginning of the string table, to the first character (not the length) of the symbol name. *name* begins on a word boundary and is preceded by a 32 bit number that contains the number of characters in the name. The symbol is terminated with an 8 bit zero, but the terminator is not included as part of the length.

This variable may point to any location within the library file (although it must always be relative to the beginning of the LST string table). In particular, it may point to a string within a symbol string table belonging to one of the SOMs contained within the library. Although this may save space in the library file, it may have a negative impact on loader performance.

If this field is not used, this symbol will be treated as unnamed common data and must be of type *storage_request*. In this case, this field will be set to 0.

Note

Zero is not a legal string table offset since the first name i string will be at offset 4.

qualifier_name

This variable is used to locate the name of a qualifier that may be user to further qualify this symbol. Its value is the byte offset, relative to the beginning of the LST string table, to the first character (not the length) of the qualifier name. The name begins on a word boundary and is preceded by a 32 bit number that contains the number of characters in the name. The name is terminated with an 8 bit zero, but the terminator is not included as part of the length.

This variable may point to any location within the library file (although it must always be relative to the beginning of the LST string table). In particular, it may point to a string within the symbol string table belonging to one of the SOMs contained within the library. Although this may save space in the library file, it may have a negative impact on loader performance.

If there is no qualifier, this field should be set to 0.

symbol_info

This field contains variant information depending on the scope of the symbol.

See “Symbol Table” on page 80.

For MPE/iX, this field can contain new values if the *shared_data* bit in the LST SOM Auxiliary header is set:

For Data Universals, this field is set to the index of the subspace the symbol is defined in.

For Storage Universals, this field is set to the size of the storage request.

For Data Unsats, this field is set to the DXRT offset for the symbol (will be a negative offset).

symbol_value

This field contains the 32 bit value of this particular symbol. Depending on the type and scope of the symbol this field may have a different meaning.

See “Symbol Table” on page 80.

For MPE/iX, this field can contain new values if the `shared_data` bit in the LST SOM Auxiliary header is set:

For Data and Storage Universals, this field is set to the DP-positive address of the symbol.

For Data Unsats, this field is set to the index of the subspace the symbol was referenced in.

symbol_descriptor

This is an argument descriptor for the procedure's type or the data type depending upon the type of the symbol (see argument descriptor definition, section 9.5).

See “Symbol Table” on page 80.

reserved

Bits 0-7

These bits are reserved for future expansion.

max_num_args

Bits 8-15

If *check_level* indicates that the number of arguments passed should be checked, the `num_args` field of the imported symbol (this field is in the exported symbol) must be less than or equal to this value.

See “Symbol Table” on page 80.

min_num_args

Bits 16-23

If *check_level* indicates that the number of arguments passed should be checked, the `num_args` field of the imported symbol (this field is in the exported symbol) must be greater than or equal to this value.

See “Symbol Table” on page 80.

num_args

Bits 24-31

This value is the number of arguments associated with the symbol.

Note

A procedure return value is NOT counted as an argument. The range of this variable is 0 to 255. Since this variable is not essential for linking or loading, compilers are not constrained to limit the number of parameters to 255. However, if this limit is exceeded, functions that use this field (e.g. parameter checking) may produce

som_index

This value is an index that identifies the SOM that defines this symbol. The index can be used (when multiplied by the entry size) to index into the SOM pointer table that follows LST header and thereby, be used to locate the SOM.

The SOM index must be a number between 0 and value of the variable *module_limit*-1 in the LST header.

This field is not used if the symbol is an import.

symbol_key

This is the 4 byte hash key for this symbol. The key is supplied to provide a quick check before comparing each byte of the symbol to determine if this is the correct symbol. Refer to “Symbol Directory” on page 104 for the hash algorithm to get this key.

next_entry

This value is the LST relative byte offset to the next entry in the list that contains this symbol. If this symbol is the last entry in the list, this field is set to zero.

Argument Descriptor Fields

See “Symbol Table” on page 80.

4.3.2 SOM Directory

The SOM directory is a table of entries that contain the location and length of every SOM within the file. Both the location and length are in bytes. The location is relative to the start of the file (not to the LST header), and points to the first byte of the SOM header (not to the archive header). The length does not include the archive header. The index of a SOM is used to index into the SOM directory.

Since each SOM will require a SOM directory entry, the variable *module_limit* in the LST header will contain the number of entries in the SOM directory. The table is pointed to by the variable *dir_loc*, which contains the LST header relative byte offset to the beginning of the SOM directory.

If a SOM does not exist, its entry in the SOM directory table will be set with a length of zero and the location set so that all bits are one.

Figure 2-25 shows the structure of the SOM directory.

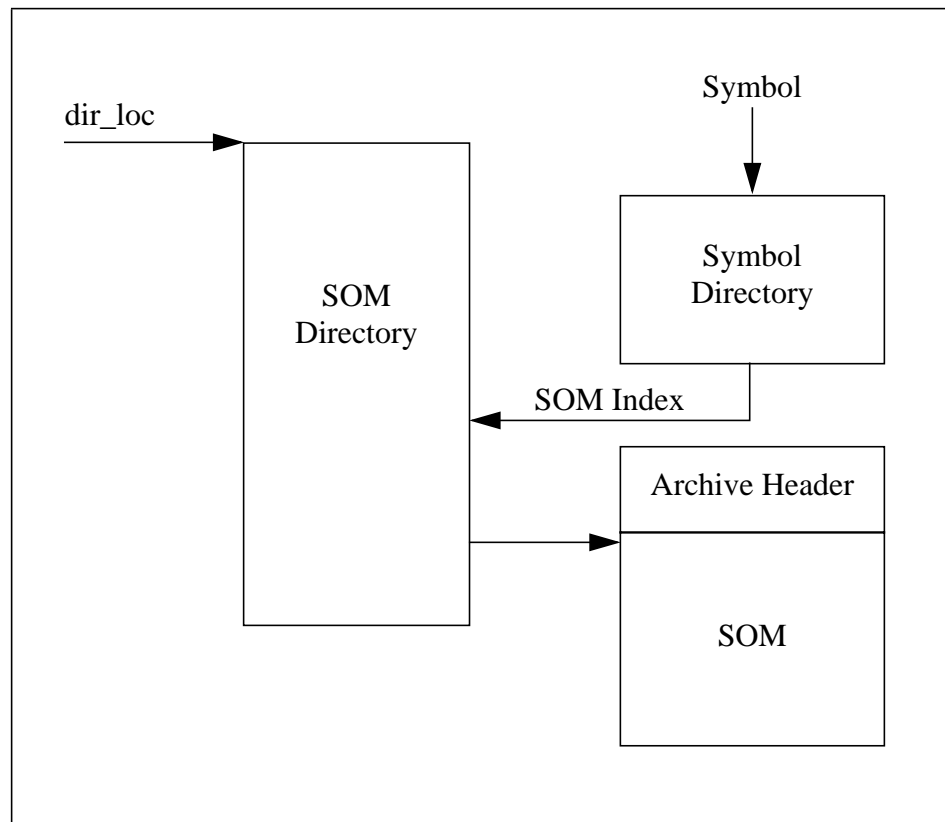


Figure 2-25: Structure of the SOM directory

4.3.3 Free Space List

A linked list of free areas within the file is maintained to support additions and deletions to an existing library file. The first free area is located by the variable *free_list*, which contains the LST header relative byte offset to the first free area. Free areas are kept track of by linking each area with a free link.

A free link is a three word link stored at the front of each free area. The first word is the LST header relative byte offset to the previous entry in the list, the second word is the

LST header relative byte offset to the next entry in the list, and the third word is the size of the current entry in bytes.

The previous link field in the first free link and the next link field in the last free link will be set to zero in order to mark the corresponding end of the free list. If the file has no free space, the free list pointer in the LST header will be set to zero.

Free space is always allocated from the free list in multiples of 4 bytes, beginning on a 4 byte boundary. If a free area is smaller than a free link it will be ignored and become lost space.

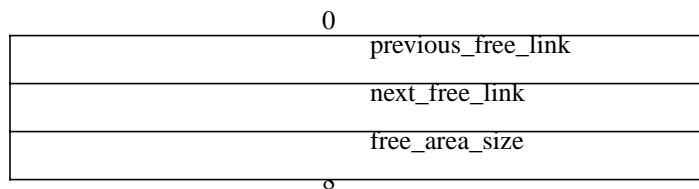


Figure 2-26: FREE_LINK Format

5.1 Object File Header

The object file header must be present, and must be at the beginning of an executable file. Magic numbers reserve for executable file are 0x107, 0x108, and most commonly 0x10B. Refer to Section 3.1 on page 43 for a detail description of the object file header and Table 10 on page 45 for more information regarding the magic numbers.

5.2 Auxiliary Headers

If the auxiliary area is present it will contain one or more auxiliary header records. The first two words of every auxiliary header record (also known as *auxiliary header identifier*) will identify the type and length of the auxiliary header. A provision has been made to allow user defined auxiliary header records, however, there will be no centralized control over the assignment of user defined auxiliary header types.

The structure of the auxiliary header id defined below in Figure 2-27.

```
struct aux_id {  
    unsigned int mandatory : 1;  
    unsigned int copy : 1;  
    unsigned int append : 1;  
    unsigned int ignore : 1;  
    unsigned int reserved : 12;  
    unsigned int type : 16;  
    unsigned int length;  
};
```

Figure 2-27: Definition of the Auxiliary Header

mandatory

Bits 0

If this bit flag is set to one it indicates that this auxiliary header contains information that the linker must understand. If the type field is undefined for the version of the linker being used, it is an error.

copy

Bits 1

If this bit flag is set to one it indicates that this auxiliary header is to be copied without modification to any new SOM created from this SOM. Two auxiliary headers with the same type field should not be merged together but left as separate entries.

append

Bits 2

This bit flag is the same as the *copy* flag above except that multiple entries with the same type and append set of “action flags” (i.e., *mandatory*, *copy*, *append*, *ignore*) should be merged (concatenation of the data portion). The order of merging is not important.

ignore

Bits 3

If this bit flag is set to one it indicates that this auxiliary header should be ignored if its type field is unknown (i.e., do not copy, do not merge).

reserved

Bits 4-15

These bits are reserved for future use.

type

Bits 16-31

This field is a numeric value that defines the contents of the auxiliary header.

This field has a range of 0 to 65535. TYPE values less than 32767 are reserved for Hewlett-Packard defined auxiliary header record types. TYPE values greater than 32767 are user definable.

The currently defined auxiliary header type values are defined in Table 7 on page 119.

TABLE 7

Auxiliary Header Types

Value	Usage
0	NULL
1	Linker footprint
2	Obsolete (used to be MEP/iX program)
3	Debugger footprint
4	Exec Auxiliary Header
5	IPL auxiliary header
6	Version string
7	MPE/iX program
8	MPE/iX SOM
9	Copyright
10	Shared Library version information
11	Product specifics
12	NetWare Loadable Module

length

This is the length of the auxiliary header in bytes. This value does NOT include the two word identifier at the front of the header.

An auxiliary header is not constrained to be an integral number of words in length. If it is not word aligned, the next auxiliary header or the end of the auxiliary header area will be placed at the next word boundary. The value of pad bytes are not defined. If two auxiliary headers are merged and the first is not word aligned, the next one will start on the very next byte.

Note

The mandatory, copy, append and ignore bits fields in the auxiliary header are not used consistently. Thus, users should consider these fields meaningless and unreliable.

5.2.1 Loader Auxiliary Headers

Currently there are three type of loader auxiliary headers:

- *HP-UX auxiliary header*: This auxiliary header contains run-time information used by the HP-UX loader to do a fast and efficient program load of an executable SOM. See Section 6.1 on page 125 for the detail structure of this auxiliary header.
- *MPE/iX program and SOM auxiliary headers*: These are auxiliary headers used by the MPE/iX loader to load program (executable SOM) or executable library.
- *IPL auxiliary header*: This auxiliary header is used to provide information that is needed for loading bootable utilities. All bootable utilities accessible through the LIF directory must have enough of a common format for IPL to load and launch utilities through a standard method. IPL may need to know the intended physical destination address for which the module was linked, as well as the entry point and the length of the image. This auxiliary header meets IPL's needs for loading and launching bootable utilities.

Following is the IPL auxiliary header definition and its fields description:

```
struct ipl_aux_hdr {  
    struct aux_id header_id;  
    unsigned int file_length;  
    unsigned int address_dest;  
    unsigned int entry_offset;  
    unsigned int bss_size;  
    unsigned int checksum;  
};
```

Figure 2-28: Definition of the Auxiliary Header

header_id

This is the auxiliary header id for an IPL SOM. The type field of this id must be 5.

file_length

This field contains the length of the entire SOM including all headers.

address_dest

This field specifies the destination address at which the file should be loaded. For those utilities which are position independent, this field can be set to -1 and IPL will load it at the first available memory after IPL.

entry_offset

This field contains the file offset of the entry point relative to the beginning of the file.

bss_size

This field specifies the length of the un-initialized data area for the program. The loader must allocate this area immediately following the initialized data and fills it with zeroes.

checksum

This field contains the checksum of the entire file. The checksum is computed as the negated arithmetic sum of every word in the file (not including itself). In other words, the arithmetic sum of all the words in a valid file, including the checksum would be zero.

5.2.2 Other Auxiliary Headers

a.Linker footprint

The linker footprint auxiliary header is used to record the last time the linker modified this SOM or LST (whichever applies). The presence of the linker footprint is optional. Following is the linker footprint auxiliary header definition and its fields description:

```
struct linker_footprint {
    struct aux_id header_id;
    char product_id[12];
    char version_id[8];
    struct sys_clock htime;
};
```

Figure 2-29: Definition of the Linker Footprint Auxiliary Header

header_id

This is the auxiliary header id for the linker footprint. The type field of this id must be 1.

product_id

bits 0--95

This twelve character array contains the HP product identification number of the linker that last modified this SOM or LST.

version_id

This twelve character array contains the HP version number of the linker that last modified this SOM or LST.

htime

The htime is a 64 bit value that represents the time the file was last modified by the linker. The htime is actually composed of two 32 bit quantities where the first 32 bits is the second of the century (maximum value is 3162240000-1, which requires 32 bits to represent) and the second 32 bits is the nano second of the second (which requires 30 bits to represent).

This value is independent of any modification time maintained by other subsystems (e.g. the file system). The use of this field is optional, but if it is not used it will be set to zero.

b.Debugger_footprint

The debugger footprint auxiliary header is used to record the last time the debugger modified this SOM or LST (whichever applies). The presence of the debugger footprint is optional. Following is the debugger footprint auxiliary header definition and its fields description:

```
struct debugger_footprint {  
    struct aux_id header_id;  
    char debugger_product_id[12];  
    char debugger_version_id[8];  
    struct sys_clock debug_time;  
};
```

Figure 2-30: Definition of the Debugger Footprint Auxiliary Header

header_id

This is the auxiliary header id for the debugger footprint. The type field of this id must be 3.

debugger_product_id

bits 0--95

This twelve character array contains the HP product identification number of the debug program that last modified this SOM or LST.

debugger_version_id

This eight character array contains the HP version number of the linker that last modified this SOM or LST.

debug_time

The *debug_time* is a 64 bit value that represents the time the file was last modified by the debugger. The *debug_time* is actually composed of two 32 bit quantities where the first 32 bits is the second of the century (maximum value is 3162240000-1, which requires 32 bits to represent) and the second 32 bits is the nano second of the second (which requires 30 bits to represent).

This value is independent of any modification time maintained by other subsystems (e.g. the file system). The use of this field is optional, but if it is not used it will be set to zero.

c. Version String Auxiliary Header

The *Version_String* auxiliary header can be used for any user-defined string. The length of the string is essentially unbounded. The string must be null-terminated. The *string_length* field contains the length of the user-defined version string, not including the null () terminator. (Note that the length field in *aux_header_id* includes both the *string_length* field and the padding bytes of the string.)

Additional auxiliary header types for other kinds of user strings may be added in the future, rather than reserving one auxiliary header type for all such user strings.

Following is the version string auxiliary header definition. Note that the type of the *header_id* field must be 6:

5.3 Symbol Table

The symbol table (also known as symbol dictionary) has the same format as the symbol table in the relocatable object file. Please refer to Section 3.7 on page 80 for detail descriptions of the symbol table.

```
struct version_string_aux_hdr {  
    struct aux_id header_id;  
    unsigned int string_length;  
    char user_string[1];  
};
```

Figure 2-31: Definition of the Version String Auxiliary Header

5.4 Stack Unwind Table

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.

Please refer to the “Stack Unwind Library” chapter for more information on unwind.

5.5 Recover Table

The recover table has three words entries that contains the beginning and the end addresses of the unwind region and the resume address. Please refer to the “Stack Unwind Library” chapter for more information on unwind and recover.

5.6 Auxiliary Unwind Table

The auxiliary unwind table is implemented to mainly support the Ada trace back mechanism. The auxiliary unwind table parallels to the unwind table and contains four words entries that describe information on the compilation unit, the scope name, the scope type and the address of the line table.. Please refer to the “Stack Unwind Library” chapter for more information on Ada procedure trace back tables and mechanism.

6.1 HP-UX Auxiliary Header

The exec auxiliary header (also known as the 'HP-UX' auxiliary header within Hewlett-Packard) is used to contain run-time information for executable SOM files which conform to the notion of a 32-bit local address space. This header is filled in by the linker and is used by the system loader. The exec auxiliary header must immediately follow the SOM header record. This auxiliary header contains all the information needed by the system loader to perform fast and efficient program load of an executable SOM. All fields are mandatory and are expected to be filled in by the linker.

The Exec Auxiliary Header is required in all incomplete executables and relocatable objects. Figure 2-32 on page 126 defines the Exec Auxiliary Header.

som_auxhdr

This field contains the auxiliary header identifier for a program file. The type field of this header id must be 4.

exec_tsize

This field specifies the text (code) size in bytes (does not have to be a multiple of 4 Kbytes). The actual size of the text section in the file must be a multiple of 4 Kbytes and can be padded with zeroes to make it a multiple of 4 Kbytes.

exec_tmem

```
struct som_exec_auxhdr {
    struct aux_id som_auxhdr; /* som auxiliary header */
    long exec_tsize;          /* text size in bytes */
    long exec_tmem; /* offset of text in memory */
    long exec_tfile; /* location of text in file */
    long exec_dsize; /* initialized data */
    long exec_dmem; /* offset of data in memory */
    long exec_dfile; /* location of data in file */
    long exec_bsize; /* uninitialized data (bss) */
    long exec_entry; /* offset of entrypoint */
    long exec_flags; /* loader flags */
    long exec_bfill; /* bss initialization value */
};
```

Figure 2-32: Definition of Exec Auxiliary Header

This field specifies the space-relative byte offset of text (code) in memory. The address must be page aligned.

exec_tfile

This field contains the location of the text (code) in the file. The value will be a byte offset relative to the first byte of the SOM.

exec_dsize

This field specifies the size in bytes of the initialized data (does not have to be a multiple of 4 Kbytes). The actual size of the data section in the file must be a multiple of 4 Kbytes and can be padded with zeroes to make it a multiple of 4 Kbytes.

exec_dmem

This field specifies the space-relative byte offset of data in memory. The address must be 4 Kbyte aligned.

exec_dfile

This field contains a location of the data in the file. The value is a byte offset relative to the beginning of the SOM.

exec_bsize

This field contains the size in bytes of the uninitialized data in the file.

exec_entry

This field contains the space-relative byte offset of the main entry point for this file.

exec_flags

This field contains a series of one-bit flags for use by the loader.

The low-order bit (bit 31) is defined to indicate whether nil-pointer dereferences should be trapped by the operating system. If the bit is set, dereferences of nil pointers will be trapped; if the bit is not set, dereferences of nil pointers will return 0.

Bit 30 indicates that external millicode (if implemented) is used by this program file.

Bit 29 indicates dynamically linked (incomplete) executables (for example, an executable linked with shared libraries).

Bit 28 indicates executable built with the aid of profile information.

Bit 24-27 - instruction page size field :

0:4 K 1:16 K 2: 64 K 4: 1M 5: 4M 6: 16 M 7: 64 M 8: 256 M

Bit 22-23 - initialized to 0 and reserved.

Bit 21 - static branch prediction recommended for this load image.

Bit 17-20 - data page size field; values as for instruction page size field above.

Bit 15 - Enable lazy swap when set.

Bit 14 - Lock text into physical memory when set.

Bit 13 - Lock data into physical memory when set.

exec_bfill

This field specifies the value to which uninitialized data (BSS) should be initialized.

6.2 Program Startup

All programs must be linked with the relocatable startup object *crt0.o*. This object code defines entry points, sets up data pointer register (DP), initializes program variables, and checks for dynamic (shared) libraries.

Table 7 on page 39 summaries program variables that are defined by *crt0*, and Table 8 on page 40 lists the register definition at process initialization.

Here is how shared libraries work at run time: Startup code in */usr/ccs/lib/crt0.o* invokes the dynamic loader, */usr/lib/dld.sl*, which in turn maps all the shared libraries, binds all the symbols, and applies all the dynamic relocations, then branches back to the invoking executable file.

The magic numbers and aux headers are the same between an incomplete and a fully bound executable. To decide whether to invoke the dynamic loader, *crt0* looks at the first word in the TEXT space, found by looking at `__text_start` symbol created by the linker. If this matches the value of the `DL_HEADER_VERSION_ID` or `DL_HEADER_VERSION_ID2` in *<shl.h>*, then the executable is an incompletely bound program file.

crt0 will then map *dld.sl*. First it opens the file; then it reads the text, data, and bss sizes from the HP-UX aux header; then it calls `mmap(2)` to map all three sections into memory. Finally, it invokes the entry point for *dld.sl* indirectly, by adding the `exec_entry` field of the aux header to the mapped address of *dld*'s text start, and makes an indirect function call to this point.

Several parameters are sent to *dld*'s main entry point in this call. These include the starting and ending addresses of *dld*'s text, data, and bss, as well as the name of the program file, its starting and ending addresses, and a value that *dld* will use as its stack location.

Since *dld* runs before the program file itself, and it uses the stack as pointed to by `%r30` for local variables just as any code, by the time the program file routines are entered, the stack is likely to be non-zero -- that is, dirty. This has caused no small concern to various (poorly-written) applications that expect their local variables to start out with a zero initial value. In order to prevent a dirty stack on program entry, *dld* bumps the stack by 8K bytes on entry, and uses this value as its starting stack address.

Actually, *crt0* bumps the stack by this new value before it invokes *dld*, then restores the original SP value upon *dld*'s return. But this stack location is sent to *dld* so that it can use the same minimum address when it is invoked during program execution: specifically, when deferred binding is on and a procedure call must be bound at first invocation.

6.2.1 Sample Assembly Listing of *crt0* code

```
#define etext _etext
```

```
#define monitor _monitor
```

```
;;; these constants come from /usr/include/shl.h
```

DL_HDR_VERSION_ID .equ 89060912

DL_HDR_VERSION_ID2 .equ 93092112

#include <machine/break.h>

.space\$TEXT\$

.subspa\$UNWIND_START\$,QUAD=0,ALIGN=8,ACCESS=0x2c,SORT=56

.subspa\$UNWIND\$MILLICODE\$,QUAD=0,ALIGN=8,ACCESS=0x2c,SORT=62

.subspa\$CODE\$

.import __text_start, data

.proc

.callinfo SAVE_SP,FRAME=128

.export\$START\$,entry

.entry

\$START\$

ldil L'\$global\$,dp ;Initialize the global data

ldo R'\$global\$(dp),dp ; pointer

ldo 128(sp),sp ;Allocate frame, marker, and argument

depi 0,31,3,sp ; list and doubleword align sp

ldw 0(arg1),r3 ;Get argv[0]...

addil L'\$ARGV-\$global\$,dp

stw r3,R'\$ARGV-\$global\$(r1) ; and stash it away

addilL'_environ-\$global\$,dp ; Initialize _environ

stwar2,R'_environ-\$global\$(r1) ; so getenv(3) works

; Floating point status register initialization.

; We use the dld ltptr location at dp-4 as a scratch area

fstws fr0,-4(0,dp)

ldil LR'_fp_status,r5 ;symbol value set by ld +FP option

ldo RR'_fp_status(r5),r5 ;default value is 0

ldw -4(dp),r4

orr4,r5,r5 ;we OR into the current status

stwr5,-4(dp) ;store result into scratch area

addil L' DL_HDR_VERSION_ID2, %r0 ; load constant here to avoid

ldo R' DL_HDR_VERSION_ID2 (%r1), %r19 ; interlock from store

fldws-4(dp),fr0; load the fp status register from the

; scratch area we saved it in

; This is the documentation of the structure pointed to by

; %arg3. Learn it, know it, live it. Its definition lives in

; /usr/include/machine/cpu.h, the structure name is keybit_info.

; +-----+

; | cpu_version |

; +-----+

; | FP status reg after copr 0,0 |

; +-----+

```

; | number of words of keybits |
; +-----+
; | Keybits_1 |
; +-----+
; | additional opt. keybits |
; |. |
; |. |
; |. |
; +-----+
; | -1 |
; +-----+

; The -1 marks the end of the list. This way, we can extend this
; structure in the future, and add fields besides keybits if we want.

.import _is_89_0

copy %r26, %r4 ; save arvc, argv and envp

copy %r25, %r5

copy %r24, %r6

copy %r23, %r7 ; save keybits pointer

ldil L'_is_89_0,r31 ; Make sure we are not on a 8.0 or 9.0 system
ble R'_is_89_0(sr4,r31) ; before we de-reference the keybits pointer

copy r31,rp

copy %r4, %r26 ; restore arvc, argv and envp

copy %r5, %r25

copy %r6, %r24

copy %r7, %r23 ; restore keybits pointer

```

comb,<>,n %ret0, %r0, L\$0002 ; If return value is non-zero, we are

; on a 9.0 system, and should not

; de-reference the keybits pointer.

; Even though we check to make sure we are not on 8.0 or 9.0

; above, still validate pointer in case we are running on a pre-8.0

; system.

; The pointer validation assumes that exec will always place

; the CPU_INFO structure higher on the stack than envp.

comb,>>,n %arg3,%sp,L\$0002 ; If passed_ptr > sp, it must be a

; bogus pointer.

comb,<<,n %arg3,%arg2,L\$0002 ; If passed_ptr < envp, it must be a

; bogus pointer.

; We have a valid pointer

ldw (%arg3),%r5 ; Get first word of structure (cpu_version)

addil L'_CPU_REVISION-\$global\$,dp ; Store _CPU_REVISION info

stw %r5,R'_CPU_REVISION-\$global\$(r1) ; passed in from the kernel

ldw 12(%arg3),%r5 ; Get first word of keybits (key_bits[0])

addil L'_CPU_KEYBITS_1-\$global\$,dp ; Store key bits loaded from stack

stw %r5,R'_CPU_KEYBITS_1-\$global\$(r1) ; into _CPU_KEYBITS_1

; Now we set up the _FPU_MODEL and _FPU_REVISION globals with

```
; the data from the fpu_info field of the keybit_info structure

ldil L'_FPU_MODEL,%r4          ; put the address of _FPU_MODEL

ldo R'_FPU_MODEL(%r4),%r4      ; in %r4

ldw 4(%arg3),%r5              ; put copr 0,0 results into %r5

extru %r5,15,6,%r6            ; put the fpu model into %r6

sth %r6,(%r4)                 ; store the fpu model in _FPU_MODEL

extru %r5,20,5,%r6            ; put the fpu revision into %r6

sth %r6,2(%r4)                ; store the revision _FPU_REVISION

L$0002

; Shared Library support -- mapping dld.sl

; check a.out file for dl_header

; dl_header is the first thing in the text space.

ldil L'__text_start,r1        ; dl_header.hdr_version

ldw R'__text_start(r1),r31

addil L'DL_HDR_VERSION_ID,%r0 ; start loading old version number

combt,=,n %r19,%r31,L$0004    ; if new version, go map dld now

ldo R'DL_HDR_VERSION_ID (%r1), %r19

combf,=,n %r19,%r31,L$0001    ; if not old version, skip mapping

L$0004

.import __map_dld

.import __stack_zero, absolute

; map_dld
```

```
; set sp to skip nominal 8K to maintain clean stack (dld uses sp+8k
; for sp) - actually linker-set value of “__stack_zero”, setable
; with ld -FS <val>, where val is in decimal bytes.

copy sp, %r7 ; save sp

addil LR' __stack_zero, sp

ldo RR' __stack_zero(%r1),sp

copy %r26, %r4          ; save arvc, argv and envp

copy %r25, %r5

copy %r24, %r6

copy sp, arg1           ;pass dld's sp as 2nd arg

;envp is already in place for 3rd arg

copy %r7, arg3          ;pass in orig user sp (saved in gr7) as 4th arg

copy r3, arg0           ;pass in program file name as 1st arg

ldil L' __map_dld,r31

ble R' __map_dld(sr4,r31)

copy r31,rp


copy %r4, %r26          ; restore arvc, argv and envp

copy %r5, %r25

copy %r6, %r24

copy %r7, sp           ; restore original sp.


L$0001


.import _start
```

```
.call

stw r0,-4(sp)           ;Mark last stack frame (null fm_psp)

addil L'_environ-$global$,dp    ; Pass in the (possibly)
ldw R'_environ-$global$(r1),arg2 ; updated value of _environ
ldil L'_start,r31
ble R'_start(sr4,r31)
copy r31,rp
$START_RTNS$
break BI1_AZURE,BI2_AZURE_CRT0  ;Should never get here
.procend

.proc                    ; so a profiling SOM will load with this.
.callinfo               ;
.export _mcount,entry   ;
.entry
_mcount
.exit
bv,n(rp)
nop
.procend

.proc
.callinfo
.export _clear_counters,entry
```

.entry

_clear_counters

.exit

bv,n (rp)

nop

.procend

.proc ; _sr4export serves as target of calls

.callinfo export_stub ; from dynamically-loaded code to the

.export _sr4export,code ; basis code.

_sr4export

ble 0(sr4,r22) ; branch to real entry point

copy r31,rp ; ...return link in rp

ldw -24(sp),rp ; restore return link from stack

ldsid (rp),r1; get space id for return

mtsp r1,sr0

be,n 0(sr0,rp); return

nop

.procend

.proc ; __d_trap is used by HP/PAK

.callinfo

.export __d_trap,entry

.entry

__d_trap

.exit

bv,n(rp) ; just return

nop

.procend

.subspa \$UNWIND_START\$;Declare subspace start symbols

.export \$UNWIND_START, data

\$UNWIND_START

.subspa \$UNWIND_END\$,QUAD=0,ALIGN=8,ACCESS=0x2c,SORT=72

.export \$UNWIND_END, data

\$UNWIND_END

.subspa \$RECOVER_START\$,QUAD=0,ALIGN=4,ACCESS=0x2c,SORT=73

.export \$RECOVER_START, data

\$RECOVER_START

.subspa \$RECOVER\$MILLICODE\$,QUAD=0,ALIGN=4,ACCESS=0x2c,SORT=78

.subspa \$RECOVER\$,QUAD=0,ALIGN=4,ACCESS=0x2c,SORT=80

.subspa \$RECOVER_END\$,QUAD=0,ALIGN=4,ACCESS=0x2c,SORT=88

.export \$RECOVER_END, data

\$RECOVER_END

.space \$PRIVATE\$

.subspa \$GLOBAL\$

.export \$global\$

.export __dld_flags, data

.export __dld_hook, data

.export __dld_list, data

; NOTE: We must always make sure that \$global\$ is double-word aligned

__dld_list

.WORD 0 ; holds address of pointer to dld library handle list

; provided to support core file debugging.

__dld_hook

.WORD 0 ; word to hold plabel of routine for dld to call back

; into the a.out so xdb can hit known breakpoint.

__dld_flags

.WORD 4 ; Bit vector for xdb or other external process to

; pass flags to crt0.o/dld.sl. All writes to

; this flag must OR in the previous contents.

;

; Meaning of bits

; 0 - if set -> dld should map libraries private

; 1 - if set -> dld should call hook routine

; 2 - if set -> dld allowed to store address

; of pointer to dld library handle list at

; location __dld_list

; 3 - if set and hook routine valid, dld should call hook routine during BOR (bind on reference).

; Dynamically changed.

```

;
; All other bits must be zero until defined later.

.WORD 0      ; leave word at dp-4 to hold LT-pointer of dld.sl
; This location is also used as a scratch area
; by startup code in crt0
;
$global$      ;Contents of dp for HP-UX

;DO NOT PUT ANY DATA ON THIS SIDE OF $global$ - YOU WILL FOUL UP
PASCAL's

;SCHEME FOR ALLOCATING THEIR MAIN PROGRAM GLOBALS HERE
INSTEAD OF ON THE

;STACK

; Define data sym to hold the system id of final executable
; __SYSTEM_ID will be defined by ld(1)

.subspa $DATA$

.import __SYSTEM_ID,ABSOLUTE

.align 8

__SYSTEM_ID

.word __SYSTEM_ID

.export __SYSTEM_ID

_FPU_MODEL
```

.half 0

; YOU MUST KEEP _FPU_REVISION IMMEDIATELY AFTER _FPU_MODEL,
SINCE THE

; CODE IN CRT0.S RELIES ON THIS!!!!!!!!!!!!

_FPU_REVISION

.half 0

.export _FPU_MODEL,data

.export _FPU_REVISION,data

_CPU_REVISION

.word 0

.export _CPU_REVISION,data

_CPU_KEYBITS_1

.word 0

.export _CPU_KEYBITS_1,data

_environ

environ

.word 0

.export _environ,data

.export environ,data,sec_def

.import _fp_status,ABSOLUTE

__d_trap_fptr

```
.word P'__d_trap

.export __d_trap_fptr,data


.subspa $PFA_COUNTER$,QUAD=1,ALIGN=4,ACCESS=0x1f,SORT=8

.export $ARGV

$ARGV .word 0;Copy of argv[0]

.align8

.export $PFA_C_START

$PFA_C_START

.subspa $PFA_COUNTER_END$,QUAD=1,ALIGN=4,ACCESS=0x1f,SORT=10

.export $PFA_C_END

$PFA_C_END


.end
```

6.3 Shared Libraries

6.3.1 Shared Library Memory Model

The HP-UX shared memory starts at hexadecimal address 0x80000000 and ends at 0xFFFFFFFF (third and fourth quadrant) with the upper 256 megabytes of the fourth quadrant reserved for system use. The above address range is mapped into memory using space register SR6 and SR7.

6.3.2 Linkage Table

The Linkage Table is located in the \$DATA\$ space of a shared library and/or program file. It is divided into two parts: a Data Linkage Table (DLT) for data references and a Procedure Linkage Table (PLT) for procedure calls. The linkage table is used as a branch table to handle indirect procedure and data references. The DLT contains an entry for each data or procedure symbol that is accessed via the DLT_REL fixup request. The

PLT contains an entry for each unresolved procedure symbol referenced within the object.

6.3.2.1 Data Linkage Table

Each DLT entry is a single word which contains a pointer to the actual data item referenced via a T' fixup; this pointer value is assigned by the dynamic loader, after mapping the shared library. Since T' references to data items go directly through the DLT (rather than a stub), the register r19 is reserved to point to the middle of the DLT, to provide maximum addressability for short load instructions. The linker allocates r19-relative offsets for each DLT entry, and uses those offsets when rewriting code that accesses data with the DLT_REL fixup.

6.3.2.2 Procedure Linkage Table

The Procedure Linkage Table (PLT) is created for both shared libraries and incomplete executables, and is placed immediately following the DLT (if one exists). A PLT entry is created for each unique procedure symbol imported by the object. The linker creates an import stub for each unresolved procedure and redirects the reference to the import stub created, which uses the address in the PLT entry to branch to the actual procedure. In PIC code (shared libraries), import stubs use a long r19-relative offset to access an entry in the PLT; therefore, PLT entries are not constrained to be a fixed distance from r19 (as the DLT references are). For non-PIC code (incomplete executables), register r19 is not reserved, and import stubs will be able to access the PLT entries directly (because the executable program knows where its Linkage Table is allocated: right before DP). Each PLT entry consists of two words: the first word contains the address of the target procedure, and the second word contains the r19 (linkage table pointer) value required by the procedure being called.

```
struct PLT_entry {  
    int proc_addr;      /* address of procedure */  
    int ltptr_value;    /* value of r19 required for this procedure */  
};
```

Figure 2-33: PLT Entry Definition

proc_addr

This field contains the address of the procedure to be branched to, taken from the export table of a shared library or program file. It can also be initialized to the address of the bind on reference (BOR) dynamic loader routine that will bind the procedure upon first reference.

ltptr_value

If `proc_addr` points to the BOR routine, this holds the import index of the code symbol. Once the actual destination address has been calculated and stored in `proc_addr`, this field holds the Linkage Table pointer value for the callee routine.

6.3.3 The DL Header and Other Tables

The DL header appears in every shared library and in incomplete executables (program files linked with shared libraries--may contain unsatisfied symbols which will be satisfied at run time by the dynamic loader). It is assumed to be at offset 0 in the `$TEXT$` space. It defines fields used by the dynamic loader and various other tools when attaching the shared libraries at run time. The header contains information on the location of the export and import lists, the module table, the linkage tables, as well as the sizes of the tables.

Figure 2-34 on page 144 defines the `dl_header` structure. The followings are its fields description:

hdr_version

This field is used to denote the version of the DL header. The old value was set to the decimal number “89060912” prior to 10.0. The new value is “93092112”.

ltptr_value

This field is the data-relative offset of the Linkage Table pointer (GR 19 for shared libraries, GR 27 for incomplete executables). The linkage table pointer is used by the dynamic loader to access the Data Linkage Table and Procedure Linkage Table entries at load time so it can bind symbols and attach shared libraries. All data references and PIC code in a shared library must go indirectly through the linkage pointer.

shlib_list_loc

This field is the text-relative offset of the shared library list. The shared library list is a list of shared libraries that the given file depends on for symbol bindings. If the shared library list in a shared library is present, the shared library is said to “depend” on the libraries in the shared library list.

shlib_list_count

This field is the number of entries in the shared library list.

import_list_loc

This field is the text-relative offset of the import list. The dynamic loader searches the import list and binds each entry in the list at load time.

import_list_count

```

struct dl_header {
    int      hdr_version;           /* header version number */
    int      ltptr_value;           /* data offset of LT pointer (R19) */
    int      shlib_list_loc;        /* text offset of shlib list */
    int      shlib_list_count;      /* count of items in shlib list */
    int      import_list_loc;       /* text offset of import list */
    int      import_list_count;     /* count of items in import list */
    int      hash_table_loc;        /* text offset of export hash table */
    int      hash_table_size;       /* count of slots in export hash table */
    int      export_list_loc;       /* text offset of export list */
    int      export_list_count;     /* count of items in export list */
    int      string_table_loc;      /* text offset of string table */
    int      string_table_size;     /* length in bytes of string table */
    int      dreloc_loc;            /* text offset of dynamic reloc records */
    /*
    int      dreloc_count;          /* number of dynamic relocation
records */
    int      dlt_loc;              /* data offset of data linkage table */
    int      plt_loc;              /* data offset of procedure linkage
table */
    int      dlt_count;            /* number of dlt entries in linkage table */
    /*
    int      plt_count;            /* number of plt entries in linkage table */
    /*
    short     highwater_mark;       /* highest version number seen in lib or
in shlib list */
    short     flags;               /* various flags */
    int      export_ext_loc;        /* text offset of export extension tbl */
    int      module_loc;           /* text offset of module table */
    int      module_count;         /* number of module entries */
    int      elaborator;           /* import index of elaborator */
    int      initializer;          /* import index of initializer */
    int      embedded_path;        /* index into string table for search
path */
    /*
    int      initializer_count;     /* index must be > 0 to be valid */
    /* count of items in initializer import
list */
    int      tdsiz;               /* size of the TSD area */
    int      fastbind_list_loc;    /* text-relative offset of fastbind info */
    /*
};

```

Figure 2-34: Definition of DL Header

This field is the number of entries in the import list.

hash_table_loc

This field is the text-relative offset of the hash table.

hash_table_size

This field is the number of slots used in the hash table.

export_list_loc

This field is the text-relative offset of the export list.

export_list_count

This field is the number of export entries.

string_table_loc

This field is the text-relative offset of the string table.

string_table_size

This field is the length of the string table.

dreloc_loc

This field is the text-relative offset of the dynamic relocation records. Dynamic relocation records are built for each data location initialized with the address of a function or data item.

dreloc_count

This field is the number of dynamic relocation records generated.

dlt_loc

This field is the offset in the \$DATA\$ space of the Data Linkage Table. The Data Linkage Table consists of one word entries for each static data item that is referenced by Position Independent Code (PIC).

plt_loc

This field is the offset in the \$DATA\$ space of the Procedure Linkage Table. The Procedure Linkage Table contains entries for each unresolved procedure call in a shared

library or for calls to exported procedure symbols. The dynamic loader binds procedure symbols at run time.

dlt_count

This field is the number of entries in the DLT.

plt_count

This field is the number of entries in the PLT.

highwater_mark

Bits 0-15

The highest version number of any symbol defined in the shared library or in the set of highwater marks of the shared libraries in the shared library list. For a program file, a highwater version of each library linked with the program is recorded.

highwater_mark is used by the dynamic loader at run time to determine which shared library symbol is to be used for binding the program file's symbol reference.

flags

Bits 16-31

This field is used to specify the dl_header flags, such as to denote if initializers or elaborators have been seen in the libraries. The valid values for this field are:

```
#define ELAB_DEFINED 1    /* an elaborator has been defined for this library */

#define INIT_DEFINED 2    /* an initializer has been defined for this library */

#define SHLIB_PATH_ENABLE 4    /* allow search of SHLIB_PATH at runtime */

#define EMBED_PATH_ENABLE 8    /*allow search of embed path at runtime*/

#define SHLIB_PATH_FIRST 16    /* search SHLIB_PATH first */

#define SEARCH_ALL_STORS 32    /* search all shlibs to satisfy STOR import */

#define SHLIB_INTERNAL_NAME 64 /*shlib has an internal name, for library-level
                                versioning support*/
```

See “Library-Level Versioning” on page 164 for details about the usage of SHLIB_INTERNAL_NAME for library level versioning support.

export_ext_loc

This field is the text-relative offset of the export extension table. The export extension table contains information about a symbol such as its size, the start of the drelloc list, and a list of exports with the same value.

module_loc

This field is the text-relative offset of the module table. The module table is a structure containing information on the modules used to build the shared library. It has the information on defined and referenced symbols for each module in the table.

module_count

This field is the number of modules in the module table.

elaborator

This field holds an index into the import table if the *elab_ref* bit in the flags field is set.

initializer

This field holds an index into the import table if the *init_ref* bit in the flags field is set and the *initializer_count* field is set 0. If *initializer_count* is non-zero, then the *initializer* field will no longer contain an import index. Instead it will be an offset of the initializer import list relative to the beginning of the \$TEXT\$ space. The contents of the table will be import indexes of the specified initializers.

embedded_path

This field is an index into the shared library string table.

initializer_count

This field holds the number of initializers declared.

tdsize

This field holds size of the TSD area.

fastbind_list_loc

This field holds text-relative offset of fastbind info.

6.3.4 Version Auxiliary Header

The shared library version auxiliary header is used to record the version number of the object module. This auxiliary header is optional. The linker can use this auxiliary header to determine the version of the exported symbols within the module plus the high water mark for a shared library or incomplete executable.

```
struct shlib_version_aux_hdr {  
    struct aux_id    header_id;  
    short           version;  
};
```

Figure 2-35: Shared Library Version Auxiliary Header Definition

aux_header_id

Bits 0-63

This field contains the auxiliary header identifier for the object module.

version

Bits 0-15

This field contains the version number of the object module. The version number is represented as the number of months since January, 1990.

6.3.5 Import List

An import list is created for both incomplete executables and shared libraries. The import list is allocated in the TEXT space of the object, and consists of an array of import entries. Each import entry contains information about the symbol name, symbol type, and the shared library which defined the symbol at link time. The import list must maintain a one-to-one correspondence with the linkage table. There is an import symbol for each DLT entry in the linkage table, followed by an import symbol for each PLT entry in the linkage table.

The following is the `import_entry` data structure, which makes up the import list in incomplete executables and shared libraries.

name

Bits 0-31

This field contains an offset into the string table denoting the symbol name.

reserved2

```

struct import_entry {
    int            name;                /* offset in string table */
    short          reserved2;          /* unused */
    unsigned char  type;               /* symbol type */
    unsigned int   bypassable : 1;     /* address of code symbol
                                     not taken in shlib */
    unsigned int   is_tp_relative;     /* new field*/
    unsigned int   reserved1 : 6;     /* unused */
};

```

Figure 2-36: Import entry structure

Bits 0-15

Unused. Initialized to -1 if a shared library, 0 if an incomplete executable.

type

Bits 16-23

This field specifies the symbol type (text, data, or bss).

bypassable

Bits 24

This bit is set (1) in shared libraries for code imports which do NOT have their address taken in that shared library. Otherwise, it is 0. The bypassable bit controls a runtime optimization performed by dld.sl. This optimization bypasses export stubs for shared library imports that are satisfied by exports from a shared library (either the same library or a different library).

reserved1

Bits 25-31

These bits are reserved for future expansion (currently initialized to 0).

6.3.6 Export Table

The export table is allocated in the TEXT space of the object and is built for both shared library and incomplete executables files. The export table has an associated hash table for fast lookup; each one-word entry in the hash table contains an index into the export entry list. The next field of the export record holds the index of the next export record on the hash chain. A NIL (-1) next value terminates the list. Each entry in the export list contains information about the symbol name, symbol type, symbol address (symbol off-

set), and symbol version number. There is a separate entry for each version of a symbol. Parameter relocation information is not currently used.

```
struct misc_info {  
  
    short version;    /* months since January, 1990 */  
  
    unsigned int reserved2: 6;  
  
    unsigned int arg_reloc: 10;    /* parameter relocation bits (5*2) */  
  
}  
  
struct export_entry {  
  
    int next;    /* index of next export entry in hash chain */  
  
    int name;    /* offset within string table */  
  
    int value;    /* offset of symbol (subject to relocation) */  
  
    union {  
  
        int size;    /* storage request area size in bytes */  
  
        struct misc_info misc;    /* version, etc. N/A to storage requests */  
    } info;  
  
    unsigned char type;    /* symbol type */  
  
    unsigned int is_tp_relative : 1;    /* TLS export*/  
  
    unsigned int reserved1 : 7;    /*reserved */  
  
    short module_index;    /* index of module defining this symbol */  
  
};
```

next

Bits 0-31

This field contains an index to the next export record in the hash chain.

name

Bits 0-31

This field contains an offset into the string table denoting the symbol name.

value

Bits 0-31

This field specifies the symbol address (subject to relocation).

info

Bits 0-63

If the exported symbol is of type STORAGE, this field specifies the size of the storage request area in bytes. Otherwise, this field contains the version of the exported symbol along with argument relocation information.

type

Bits 0-7

This field specifies the symbol type. Valid symbol types are:

ST_CODE

ST_DATA

ST_STORAGE

ST_PLABEL

reserved1

Bits 8-15

These bits are reserved for future expansion.

module_index

Bits 16-31

This field contains the index into the module table of the module defining this symbol.

6.3.7 Export Table Extension

The export table extension is allocated in the TEXT space of the object and only appears in shared libraries. It runs parallel to the export table and provides extra information about each export record. Currently, the information in this extension contains information needed to perform data copying from a shared library to the program file. It indicates the size in bytes of each data item as well as any dynamic relocations that must be applied. A same list field is included to ensure that all data symbols that refer to the same physical location within the shared library are copied to the program file. This ensures that all alias names, common with secondary defs, refer to the same location in the resulting program. The information in the export extension table is only used at link time, in order to correctly apply DR_PROPAGATE dreloc records; it currently is not accessed by the dynamic loader anywhere.

```
struct export_entry_ext {  
  
    int size;      /*export symbol size, data only */  
  
    int dreloc;    /* start of dreloc list for this symbol */  
  
    int same_list; /* circular list of exports that have the same value */  
  
    int reserved2;  
  
    int reserved3;  
  
};
```

size

Bits 0-31

This field is the size in bytes of the export symbol and is only valid for exports of type ST_DATA. For other export types, this field is initialized to -1.

dreloc

Bits 0-31

This field is the start of the dreloc records for the exported symbol. If no relocation records exist for this symbol, this field is initialized to -1.

same_list

Bits 0-31

This field is a circular list of exports that have the same value (physical location) in the library. This is to ensure that all data symbols that refer to the same physical location in the library are copied to the program file.

reserved2

Bits 0-31

This field is reserved for future expansion (currently initialized to 0).

reserved3

Bits 0-31

This field is reserved for future expansion (currently initialized to 0).

6.3.8 Shared Library List

The shared library list is built for both shared libraries and incomplete executables. This list is allocated in the TEXT space, and contains an entry for each shared library specified at static link time. The shared library list is an array of entries which contain information about the library name, whether the library was specified with “-lc” or as an absolute path name, and whether the library was specified with an immediate or deferred binding attribute. The shared library name, as placed into the string table, should be the fully qualified path name of the shared library as determined at static link time. Please see Section 6.3.15 on page 164 for details of handling library versioning when the `internal_name` bit is set.

```
struct shlib_list_entry {  
  
    int shlib_name;    /* offset within string table */  
  
    unsigned char reserved1:6;  
  
    unsigned char internal_name:1;    /* shlib entry is an internal name */  
  
    unsigned char dash_l_reference:1; /*referenced with -lc or absolute path */  
  
    unsigned char bind;    /* BIND_IMMEDIATE, BIND_DEFERRED or  
                           BIND_REFERENCE */  
  
    short highwater_mark; /* highwater mark of the library */  
  
}
```

shlib_name

Bits 0-31

This field contains an index into the string table of the fully qualified path name of the shared library specified at static link time.

reserved1

Bits 0-5

This field is reserved for future use.

internal_name

Bits 6

This field is a flag to indicate if shared library entry is an internal name. Please see Section 6.3.15 on page 164 for details of handling library versioning for specifying internal name with the +h linker option.

dash_l_reference

Bits 7

This field is a flag to denote if the shared library was specified on the link line with the -l option or not. If specified with -l, this flag is set to true. If the incomplete executable was linked with either the +b or +s options, the dynamic loader will search for those libraries specified with -l at link time using the path(s) given. This allows a different path to be searched at run time than what was specified at link time.

bind

Bits 8-15

This field describes the binding-time preference specified at link time when the program is built. Valid binding modes are bind-deferred and bind-immediate. Bind-deferred means the symbols are bound upon reference by the dynamic loader. Bind-immediate means the symbols are bound at program start-up.

highwater_mark

Bits 16-31

This field contains the highwater_mark seen in the shared library at link time and is only valid for shared library lists located in program files.

6.3.9 Module Table

The module table is allocated in the TEXT space and is only present in shared libraries. This table was implemented to support the smart-bind binding algorithm within dld.sl. The table consists of records that describe the symbols that are imported from the modules (object files) that comprise the library. These records allow the loader to select which imports need to be resolved based on which modules are reachable. This is very similar to the way the linker deals with archive libraries at link time. The linker selects modules based on their ability to resolve current unsats of the main program. As these modules are selected, they introduce new unsatisfied symbols that must then be resolved. Eventually, imports are resolved without the need of more modules and we have closure for a correct program. If closure cannot be reached, unsatisfied symbol errors will result. The *drelocs* field indicates the relocation records that must be applied if this module is used. The *module_dependencies* field indicates the number of modules that this module directly depends on. Direct dependency can result when one module calls a routine in another module and these symbols are then hidden. Since there is no symbolic trace of the call, the loader cannot detect the dependency through symbol records. The *imports* field points to an array of integers used to determine dependencies. Module dependencies appear first on this list followed by *import_count* import table indices.

```
struct module_entry {

    int drelocs;    /* text offset into module dynamic relocation array. */

    int imports;    /* text offset into module import array */

    int import_count; /* number of entries into module import array */

    char flags;      /* currently flags defined: ELAB_REF */

    char reserved1;

    unsigned short module_dependencies;

    int reserved2;

}
```

drelocs

Bits 0-31

This field is a text address (subject to relocation) into the dynamic relocation table.

imports

Bits 0-31

This field contains a text address (subject to relocation) into the module import table. This table is a list of import symbols and module table indices. The modules and symbols in this list must be resolved before the module can be used.

import_count

Bits 0-31

This field is the number of import symbol entries in the module import table belonging to this module.

flags

Bits 0-7

This field denotes if an elaborator was referenced in the module

module_dependencies

Bits 8-15

This field is the number of modules the current module needs to have bound before all of its own import symbols can be bound.

reserved2

Bits 0-31

This field is reserved for future expansion (currently initialized to 0).

6.3.10 Shared Library Unwind Info

The `shlib_unwind_info` structure is used to provide the necessary unwind information for debugging shared library code. The debuggers (adb, xdb) need a way to access the unwind tables for shared libraries. The dynamic loader will also use this table to access stack unwind, try/recover and line table information. Currently, in a program file, the unwind information is accessed symbolically, using the `$UNWIND_START$`, `$UNWIND_END$`, `$RECOVER_START$`, and `$RECOVER_END$` symbols. For shared libraries, there will be separate unwind tables for each shared library at addresses which are unknown at static link time; therefore the `shlib_unwind_info` structure must be accessed through a known offset off of r19 (which is reserved to point to the Linkage Table for a shared library). The `shlib_unwind_info` structure is only placed into shared library files, since program files can continue to access the unwind information symbolically. One DLT entry at `r19 + 0`, is reserved to contain an r19-relative offset to the following structure:

```
struct shlib_unwind_info {
```

```
int magic;           /* magic number for unwind detection */

int shlib_name;      /* index into string table */

int text_start;      /* virtual address of the start of text */

int data_start;      /* virtual address of the start of data */

int unwind_start;    /* text-relative offset of unwind table */

int unwind_end;      /* text-relative offset of stub unwind table */

int recover_start;   /* text-relative offset of recover table */

int recover_end;     /* text-relative offset of the line table */

};
```

This structure is initialized by the static linker which sets the `shlib_name` field to point to the shared name of the shared library in the string table and sets the `unwind_start`, `unwind_end`, `recover_start`, and `recover_end` fields to text-relative offsets for the corresponding tables. The dynamic loader will then fill in the `text_start` and `data_start` fields when the library is mapped into memory, and the `unwind_start`, `unwind`, `recover_start` and `recover_end` fields will be patched with the virtual address for the unwind tables.

magic

Bits 0-31

This field identifies the header as a shared library unwind header.

shlib_name

Bits 0-31

This field is the name of the shared library. Within the shared library file, this field holds an offset into the shared library string table. At run time, the dynamic loader converts this offset into the actual unwind address of the string.

text_start

Bits 0-31

This field specifies the presumed virtual address of the start of data. At run time, this field is relocated to hold the true address at which data is mapped.

data_start

Bits 0-31

This field specifies the presumed virtual address of the start of data. At run time, this field is relocated to hold the true address at which data is mapped.

unwind_start

Bits 0-31

This field denotes the presumed text address of the stack unwind table. At run time, this field is relocated to hold the true unwind address of the stack unwind table.

unwind_end

Bits 0-31

This field denotes the presumed text address of the stub unwind table. At run time, this field is relocated to hold the true address of the stub unwind table.

recover_start

Bits 0-31

This field denotes the presumed text address of the start of the try-recover table. At run time, this field is relocated to hold the true address of the try-recover table.

recover_end

Bits 0-31

This field denotes the presumed text address of the line table. At run time, this field is relocated to hold the true address of the line table.

6.3.11 String Table

The string table is allocated in the TEXT space for both shared libraries and incomplete executables. This table consists of a series of null-terminated strings, which represent the names of all symbols exported or imported in this file, and all library names specified at static link time. Note: this string table is distinct from the “normal” string table in a SOM.

6.3.12 Dynamic Relocation Records

Dynamic relocation, or dreloc records are used by the dynamic loader to apply run time patches to the data area of shared libraries and incomplete executables. A dynamic relocation record is built in an object each time it has a data item initialized to the address of

a shared library's function or variable. The dynamic relocation record is needed since the linker does not know the actual address for code and data items within a shared library; the final address of library code and data is only known at run time, after the shared library has been mapped into memory. When the executable imports data, using data copying, that is affected by that library's relocation record (i.e. it imports a data item that needs relocation) a special DR_PROPAGATE relocation record is generated in the program file that allows the loader to first determine the original shared library that supplied the data item and then use the relocation records within the shared library to update the data item that has been copied to the program file. When an incomplete executable imports data from a shared library, only the data item itself is copied into the executable, with the size of the data item being determined by the export extension record.

With HP-UX 9.0, run time data copying has been implemented as well as copying the data statically at link time. This causes a DR_PROPAGATE dreloc record to be emitted for each data copied object between a shared library and the program file. (The current plan is to eliminate data copying entirely for HP-UX 10.0; this will obsolete the use of the DR_PROPAGATE dreloc record altogether.)

```
struct dreloc_record {
    int shlib;          /* Reserved */

    int symbol;         /* index into import table of shlib if *_EXT type
                        low-order 16 bits used for module index if *_INT type*/

    int location;       /* offset of location to patch data-relative */

    int value;          /* text for data-relative offset to use for patch if
                        internal-type fixup */

    unsigned char type; /* type of dreloc record */

    char reserved;      /* currently unused */

    short module_index; /* Reserved */

}
```

shlib

Bits 0-31

Reserved.

symbol

Bits 0-31

This field is an index into the import table if the relocation is an external type.

location

Bits 0-31

This field is the data-relative offset of the data item the dreloc record refers to

value

Bits 0-31

This is the text or data-relative offset to use for a patch if it is an internal fixup type.

type

Bits 0-7

The field represents the of the dynamic relocation record. Valid relocation types are:

```
#define DR_PLABEL_EXT    1  /* initialized to a external code plabel (PLT)*/  
#define DR_PLABEL_INT    2  /* initialized to internal (local code plabel (PLT)*/  
#define DR_DATA_EXT      3  /* initialized to external data symbol */  
#define DR_DATA_INT      4  /* initialized to internal data offset;  
                             data-relative "value" field */  
#define DR_PROPAGATE     5  /* data item copied from shared library into a.out*/  
#define DR_INVOKE        6  /* invoke elaborator function */  
#define DR_TEXT_INT      7  /* initialized to internal text offset; text-relative  
                             "value" field */
```

Note that DR_INVOKE is for C++ shared libraries with static constructors. A C++ shared library is built with a procedure called an “elaborator”, identified by a symbol

index in the `dl_header`. For each `DR_INVOKE` relocation record seen, the elaborator is called with three arguments, the location field from the relocation record, the symbol index from the relocation record, and the shared library handle. `DR_INVOKE` are applied after all other fixups.

The `PLABEL_EXT` relocation record is the result of an initialized function pointer in the data segment. It points to the code import list entry, which corresponds to a PLT slot. The dynamic loader will fixup the initialized function pointer with the address of the “canonical” PLT entry for the referenced procedure, which may or may not be the one provided by the importing module. Every module that creates a plabel allocates a PLT slot for the imported procedure, and the loader picks one to serve as the canonical one. This ensures that plabels for the same routine will compare equal. Unfortunately, there are still cases where this cannot work, like when libraries are dynamically loaded and unloaded. For this reason, we have a plabel comparison millicode routine that compares the contents of plabels rather than their addresses.

reserved

Bits 8-15

These bits are reserved for future expansion (currently initialized to 0)

module_index

Bits 16-31

Reserved.

6.3.13 Loading Shared Libraries

6.3.13.1 Loading Libraries

When a program begins execution, the first thing it does is attach all shared libraries that were searched at link time. This activity is performed by the startup code in `crt0.o`, which maps in the *dynamic loader* which then scans a list (built at link time and stored in the program file) of shared libraries that were searched by the linker.

This list of libraries in the program file contains the paths of the libraries specified on the linker command line. Library names referred to with the `-l` option will be expanded by the linker to the fully qualified pathname for the library, as found at link time.

If a library is listed explicitly, without the `-l` option, the library name in the list will be exactly as specified on the command line.

The directories searched by the linker are by default, */usr/lib* and */opt/langtools/lib*, but they may be overridden by the environment variable *LPATH* (see the *ld(1)* manual page for details).

Note that the *LPATH* specified at link time will be used when creating the shared library list used by the dynamic loader, that is, the shared library names will be the fully qualified path names of the libraries as found at link time. The *LPATH* environment variable will not be used during dynamic loading of the library.

6.3.13.2 Dynamic Library Path Support

On the Series 700/800, support has been added for the run-time path lookup of shared libraries needed by a program file. Directory search information can come from two sources; the program file itself and an environment variable.

The program provides directory search information if it is linked with the *+b path_list* option where *path_list* is a list of directories to search. If *path_list* is a single colon *':'*, the linker will construct a list of directories to search consisting of all the *-L* directories followed by the directories specified by the *LPATH* environment variable. The directory search list will be stored in the program file itself and will be made available to the dynamic loader at run-time.

The environment variable *SHLIB_PATH* can be used by the dynamic loader to dynamically locate shared library files if the program file was linked with the *+s* option. If both the *+b* and *+s* options are specified at link time, the relative order of these options on the command line indicates which path list will be searched first. The environment will be scanned once at program start up for the value of the *SHLIB_PATH* environment variable. Future modifications to this environment variable by the executing program will not be picked up by the dynamic loader.

If dynamic path lookup is enabled either through *+b* or *+s*, only shared libraries specified on the link line via the *-l* option are subject to path lookup. For libraries loaded via the *shl_load()* call, the library will be subject to dynamic search only if the *DYNAMIC_PATH* flag is passed to *shl_load()*.

For both the *SHLIB_PATH* environment variable and the path list specified via the *+b* option, a path list consists of a colon *':'* separated list of directories with leading and trailing colons *':'* being optional. The directories will be searched in the order in which they appear in the path list. A null directory specification *“::”* indicates that the default library path stored by the linker in the program file or provided via a *shl_load()* call should be used at that point in the search. If a directory specified in the path list is relative (does not begin with a *’/’*), the directory actually searched will depend on the current working directory, not the directory where the program file actually resides. For example, if *SHLIB_PATH* were set to the path list *“/usr/lib/X11::./mylibs:/usr/lib/Motif1.1”* and the loader was presented with a shared library path list via a *shl_load()* call or by searching the library list in the program file or the dependency list of a shared library, the following locations would be probed in order:

input library path: */mnt/usr/local/thislib.sl*

- 1) /usr/lib/X11/thislib.sl
- 2) /mnt/usr/local/thislib.sl
- 3) \$PWD/./mylibs/thislib.sl
- 4) /usr/lib/Motif1.1/thislib.sl

If the loader has attempted to perform a dynamic path lookup for a shared library and failed to find it using the supplied directories, it will search the default path list of “:”.

Note, no special provisions related to security issues are taken for programs that perform `chown(2)` or `chgrp(2)`. The builder of such a program file must ensure that the user cannot substitute his own library on a search path and gain undesirable privileges. Since the default when building the program file is to not allow any dynamic shared library searching, this is not considered a security hole in the program development environment, rather it is a responsibility of the program builder.

The `chatr(1)` command has been modified to allow the user to control several aspects of shared library behavior. The options include:

- `-B bind` - Modify symbol binding modes, same as `ld(1) -B`
- `+b flag` - Control whether the program directory path list can be used, flag = enable or disable
- `+s flag` - Control whether the environment variable `SHLIB_PATH` can be used, flag = enable or disable
- `-l library` - Indicates that the specified shared library is subject to dynamic path lookup.
- `+l library` - Indicates that the specified shared library is not subject to dynamic path lookup.

6.3.14 Intra-library Version Control

Prior to 10.0, all library versioning are done at the “intra-library” level in that version control is done at program object level. Please refer to “Programming on HP-UX” for details on how to handle version control by using compiler directives and linker options. Since code from a shared library is mapped in at run time from a separate shared library file, modifications to a shared library may alter the behavior of existing executables. In some cases, this may cause programs to operate incorrectly. A means of version control is provided to solve this problem.

Whenever an incompatible change is made to a library interface, both versions of the affected module or modules are included in the library. A mark indicating the date (month/year) the change was made is recorded in the new module in a Shared Library Version Auxiliary Header (See “Version Auxiliary Header” on page 148.) This date

applies to all symbols defined within the module. A high water mark giving the date of the latest incompatible change is recorded in the shared library, and the high water mark for each library linked with the program is recorded in the incomplete executable file.

At run time, the dynamic loader checks the high water mark of each library and loads the library only if it is at least as new as the high water mark recorded at link time. When binding symbolic references, the loader chooses the latest version of a symbol that is not later than the high water mark recorded at link time. These two checks help ensure that the version of each library interface used at run time is the same as was expected.

6.3.15 Library-Level Versioning

Starting at HP-UX 10.0, shared library versioning will now be provided on an entire library. We will refer to this as “library-level versioning”, as distinguished from “intra-library” shared library versioning we provided prior to HP-UX 10.0. Note that the intra-library versioning functionality will not be going away anytime soon, as some users depend on this functionality; the library-level scheme will be an additional feature.

Here is how library-level versioning works in general: The traditional name of a delivered shared library will now be a symbolic link that points to the latest version of that library on the file system. All the “real” shared libraries will be suffixed with the pattern

`lib_name.<digit>`

instead of “lib_name.sl”; e.g., “libc.2”. Many versions of shared libraries may reside on the system at a given time, older versions will use lower numbered digits. The internal name, e.g. “lib2.2” is recorded in the library when it is built. See Section 6.3.15.1 on page 166 for details of building libraries with internal names.

When the user links an application against a shared library on the filesystem, the file specified will have a standard “.sl” suffix; normally this is done with a “-l<name>” option to the linker, which searches for a shared library called “<path>/libname.sl”.

Since this library is a symlink to the latest version available, the linker will actually open this latest shared library and link against it; it is the internal name of *this* library that is recorded in the library list of the application. For example, if these files exist on a system (Note: at HP-UX 10.0, the highest digit suffix will be “1”):

`/usr/lib/libfoo.0`

`/usr/lib/libfoo.1`

`/usr/lib/libfoo.sl -> ./libfoo.1`

`/usr/lib/libbar.0`

`/usr/lib/libbar.1`

/usr/lib/libbar.2

/usr/lib/libbar.sl -> ./libbar.2

/usr/lib/libc.0

/usr/lib/libc.1

/usr/lib/libc.2

/usr/lib/libc.3

/usr/lib/libc.sl -> /usr/lib/libc.3

and the user links an application with this command line:

```
ld /usr/ccs/lib/crt0.o main.o -lfoo -lbar -lc -o prog
```

then these shared libraries will be recorded in the file “prog”:

/usr/lib/libfoo.1

/usr/lib/libbar.2

/usr/lib/libc.3

If then in subsequent releases all of these libraries were versioned with incompatible changes (e.g.: if “libfoo.sl” now pointed to a new library, “libfoo.2”), the file “prog”, if not relinked, would always bind against these *same* shared libraries, which would remain *completely unchanged* for the life of the application.

Libraries loaded programmatically, dynamically loaded libraries (shl_load(3x)), should explicitly load the *real* file. For example, loading /usr/lib/libfoo.1 explicitly rather than the symlink /usr/lib/libfoo.sl. In this way, when the application is moved forward, it will always use the correct version.

6.3.15.1 Building libraries

In order to use the “library-level” versioning scheme, libraries must be built with the new linker option, “+h <internal-name>”. This “internal name” to be supplied on the linker command line is usually the basename of the file where it will eventually be installed. When the +h option is specified, the SHLIB_INTERNAL_NAME in the flags field of the dl_header will be set to true. Using the same example in this section, the latest versions of the three libraries will be built as follow:

```
ld -b *.o ... +h libfoo.1 -o libfoo.1
```

```
ld -b *.o ... +h libbar.2 -o libbar.2
```

```
ld -b *.o ... +h libc.3 -o libc.3
```

This “internal name” will be used by the linker to write into the library list of any application or shared library that is linked against the symbolic links of a shared library; not the name of the file itself. The linker will use the *directory* where it searches for the library, concatenated with the *internal name* in the library, to be recorded in the library list. For example, if /usr/lib/libfoo.sl is the shared library used to link with, and this file has an internal name of “libfoo.1”, then the name recorded in the library list will be /usr/lib/libfoo.1. Users must take extra caution when a path is specified in the internal name. If the internal name is an absolute path, i.e. file name that begins with “/”, then the recorded dependency in the library list is the absolute path name. Otherwise, the base name is appended to the recorded dependency path.

For example:

If /usr/lib/libfoo.sl

is a sym link to

/xxx/libfoo.1

<u>internal name with +h</u>	<u>recorded dependency</u>
libfoo.1	/usr/lib/libfoo.1
/xxx/libfoo.1	/xxx/libfoo.1
../mylib/libfoo.1	/usr/lib/../mylib/libfoo.1

Please see “Programming on HP-UX” for details on how to build libraries with “Library-Level Versioning”.

6.3.15.2 Pre-10.0 Applications

At 10.0, the dynamic loader has an enhancement to help migrate 9.0 applications because of the filesystem changes. Any shared library in /lib will be searched for in /usr/lib before the dynamic loader aborts, so /lib/libc.sl will be found correctly in /usr/lib/libc.sl.

6.3.15.3 Migrating to Library-Level Versioning

The dynamic loader has a special “compatibility enhancement” in place, strictly for the purposes of providing a suitable run-time environment for older applications moving forward when applications are migrating to Library-Level versioning.

If the dynamic loader encounters an application with no internal name specified, then when it attempts to bind a shared library to the process it will first change the “.sl” suffix to “.0” in the filename before it makes the open(2) call. If it does not find a “.0” shared library, it will use the library as *specified in the application*.

For example, if there is no /usr/lib/libfoo.0 on the 10.0 system, it will look for and use /usr/lib/libfoo.sl, just like it did for 9.0x systems.

6.3.16 Import and Export Stubs

All procedure calls from the shared library to entry points outside the library, or to exported entry points in the library, are routed through import stubs. These stubs are created by the linker in the code space of the library, and manage both the indirect reference through the linkage table and the possibility of inter-space procedure calls.

In addition, each exported procedure (including non-exported procedures whose addresses are taken) is assigned an export stub, which handles the return path of inter-space calls.

The stubs generated by the linker are defined as follows:

Import Stub (Incomplete Executable)

X':	ADDIL	L'lt_ptr+loff,dp	; get procedure entry point.
	LDW	R'lt_ptr+loff(1),21	
	LDW	R'lt_ptr+loff+4(1),r19	; get new r19 value.
	LDSID	(r21),r1	
	MTSP	r1,sr0	
	BE	0(sr0,r21)	; branch to target.

STW rp,-24(sp) ; do this as a favor to the export stub.

Import Stub (Shared Library)

```
X':  ADDIL    L'ltoff,r19                      ; get procedure entry point.
      LDW     R'ltoff(r1),r21;
      LDW     R'ltoff+4(r1),r19               ; get new r19 value.
      LDSID   (r21),r1
      MTSP    r1,sr0
      BE      0(sr0,r21)                      ; branch to target.
      STW     rp,-24(sp)                      ; do this as a favor to the export stub.
```

Export Stub (Shared libs and Incomplete Executables)

```
X':  BL,N     X,rp                            ; trap the return.
      NOP
      LDW     -24(sp),rp                      ; restore the original rp.
      LDSID   (rp),r1
      MTSP    r1,sr0
      BE,N    0(sr0,rp)                      ; inter-space return.
```

6.4 System Calls

The HP-UX operating system defines a large set of system calls. These system calls can be made indirectly by calling the interface routines in the C run-time library, or they can be made directly from assembly code. All system calls are funneled through a single entry point in the system space, which is identified by space register 7 (SR7). Each system call is assigned a unique number, which must be loaded into general register 22 (GR22). The arguments to the system routine should be loaded into argument registers: GR26, GR25, GR24, and GR23 (arg0, arg1, arg2, arg3 respectively) as necessary. When the system call returns, a status code is also returned in GR22. If the status code is zero, the system call succeeded and the return value, if any, is in GR28. If the status is not zero, the system call failed and the error number is found in GR28.

A list of system call numbers as well as the location of the system call entry points is in the standard include file */usr/include/sys/syscall.h*.

Following is an example of a code fragment shows a call to the *read* system call:

```
READCALL
      or      %r0, %r0, %arg0               ; file descriptor = 0
      addil   L%buf-$global$, %dp          ; set up buffer address in arg1
      ldo     R%buf-$global$(%r1), %arg1
      ldo     10, %arg2                      ; length = 10 into arg2
      ldil    L%0xC0000004, %r1            ; load system call entry point and
      ble     R%0xC0000004(%sr7,%r1); branch to it
      ldo     3, %r22                       ; read system call number is 3
```

In the above code fragment, the last instruction loads the constant 3, which is the unique number for the *read* system call, into GR22, and executes in the delay slot of the BLE instruction.

The standard procedure calling convention should be used to call the system call interface routines in the C library.

Symbolic Debug Information

7.1 The Debug Information Organization

The debug information are generated by the compilers, fixed up by the linker, and used by various programs (primarily the symbolic debugger(s)) to reconstruct information about the program.

On PA-RISC, a major goal was that the linker needs not know anything about the format. To this end, it was decided that the debug information be composed of several unloadable subspaces within an unloadable space (named \$DEBUG\$), and that at link time, updates to the debug information be made through the standard mechanism of a list of fixups. The linker will perform the required fixups for the debug spaces, and subspaces from separate compilation units will be concatenated. However, at exec time, the loader would know that the debug space is not to be loaded.

The debug information consists of six tables: a header table and five special tables. The header table contains one header record for each compilation unit. Each header record identifies the size (in bytes) of the five tables generated by that compilation unit. Two of the tables are very similar. The GNTT and LNTT both contain name and type information (NTT for Name and Type Table). The GNTT contains information about globals, and is thus limited to variables, types, and constants. The LNTT is for information about locals. The LNTT must therefore contain scoping information such as procedure nesting, begin-end blocks, etc. The GNTT and LNTT are both DNTTs (Debug Name and Type Tables), so the prefix DNTT is attached to objects (like a DNTTPOINTER) that are relevant to both the GNTT and LNTT. The SLT contains information relating source (or listing) lines to code addresses. The SLT and LNTT contain pointers between the two tables, so that the scoping information contained in the LNTT can also be used with the SLT. The VT contains ascii strings (such as variable names) and the values of named constants.

The five tables are summarized below:

TABLE 8

Debug Table

Table	Abbr.	Contents	Points to
Global symbols	GNTT	Global name-and-type info.	GNTT
Local symbols	LNTT	Local name-and-type info.	GNTT, LNTT, SLT, VT
Source lines	SLT	Source / listing line info.	LNTT, SLT
Value	VT	Names and constants	
Xref	XT	File offsets and Attributes	XT, VT

The pointers needed within the debug tables are in fact indexes into the tables. The GNTT, LNTT, and SLT each consist of a series of equal-sized entries. Some GNTT entries begin a data structure and some are extension entries. Some SLT entries are “special” (point back to the LNTT), others are “assist” (point forward in the SLT), but most are “normal” (point to code). There can be pointers from the LNTT to the GNTT, as it is common to have local variables of a global type. However, there are never pointers from the GNTT to the LNTT, as global variables are never of a local type.

The tables are defined to be as machine-independent as possible, but the debugger may need to “know” some facts about the system and language it is dealing with. The GNTT and LNTT are the only tables that require fixups to be generated by the compiler and acted upon by the linker. There are other fixups to be done, but these are all done by the pre-processor.

7.2 Compilation Unit Headers

7.2.1 Basic typedef and structure definitions

The following basic typedefs and structure definition are used through out this chapter:

```
typedef long          ADDRESS;
typedef unsigned long ADRT, *pADRT;
typedef unsigned int  LANGTYPE;
typedef unsigned long STATTYPE;    /* static-type location */
typedef long          DYNTYPE;     /* dynamic-type location */
typedef unsigned long REGTYPE;     /* register-type location */
typedef unsigned int  BASETYPE;
```

```
typedef unsigned int    BITS;
```

DNTT pointer:

```
struct DNTTP_IMMEDIATE {
    BITS    extension: 1;           /* always set to 1 */
    BITS    immediate: 1;          /* always set to 1 */
    BITS    global: 1;             /* always set to 0 */
    BASETYPE type: 5;              /* immediate basetype */
    BITS    bitlength: 24;         /* immediate bitlength */
};

struct DNTTP_NONIMMED {
    BITS    extension: 1;           /* always set to 1 */
    BITS    immediate: 1;          /* always set to 0 */
    BITS    global: 1;            /* 1 => GNTT, 0 => LNTT */
    BITS    index: 29;            /* DNTT table index */
};

typedef union {
    struct DNTTP_IMMEDIATE dntti;
    struct DNTTP_NONIMMED dnttp;
    long    word;                /* for generic access */
} DNTTPOINTER;                  /* one word */
```

A DNTTPOINTER of DNTTNIL means a nil pointer. In the DNTTImmediate case there is always at least one zero bit (the globalbit) to distinguish that case from nil pointer (-1). In thenon-immediate, non-nil case DNTTPOINTER is the block index, base zero, of another DNTT entry; the global bit indicates which table it is an index into, the GNTT or LNTT. Each block is 12 bytes.

Extension bits really have nothing to do with DNTT pointers, but are needed for constructing the DNTT. See the next section.

Bitlength is the MINIMUM (packed) size of the object. In lieu of other information (i.e., outside of a structure or array), the object is assumed to be right-justified in the minium number of whole bytes required to hold the bit length.

An immediate DNTTPOINTER is only allowed if the type is a simple BASETYPE. Otherwise, a separate DNTT entry must be used.

SLT pointer:

Signed entry index, base zero, into the source line table. Each entry is eight bytes.

```
typedef long    SLTPOINTER;
```

VT pointer:

Unsigned byte offset into the value table. Note that VTNIL is not actually a nil pointer, but rather a pointer to a nil string.

```
typedef long    VTPOINTER;
```

XREF pointer:

Signed entry index, base zero, into the cross reference table. Each entry is four bytes.

```
typedef long    XREFPOINTER;  
typedef int     KINDTYPE;
```

7.2.2 XDB Header structure definition:

The header table is composed of five word header records. For each compilation unit, the compiler must generate a header record, indicating the length (in bytes) of the five tables (GNTT, LNTT, SLT, VT and XT) produced for that compilation unit.

The five tables are each contained in a separate subspace on PA-RISC and at link time, the tables from different compilation units will be concatenated separately:

GNTTs to GNTTs, SLTs to SLTs, etc.

```
struct XDB_header {  
    long    gntt_length;  
    long    lntt_length;  
    long    slt_length;  
    long    vt_length;  
    long    xt_length;  
};
```

The preprocessor requires the number of compilation units, and the size of each of the five tables produced by each compilation unit. The header records supply this size information, and the number of header records equals the number of compilation units.

In PA-RISC, the header_extension flag (MSB) is set in the gntt_length word in each header-record by the compilers to indicate the header contains an xt_length and is words long. This bit is used to distinguish SOM's that were created with the earlier version of compilers which do not have an \$XT\$ subspace.

7.3 Name and Type Tables

The DNTT consists of a series of three-word blocks. Each starts with an "extension bit". Each structure in the union "dnttentry" begins in an "initial block" with a bit which is always zero. If a structure is more than three words (one block) long, it occupies one or more additional "extension blocks", each of which starts with a bit set to one to distinguish it from an initial block.

Note that every DNTTPOINTER has a high bit of one and that every DNTT structure bigger than one block is carefully arranged so that a DNTTPOINTER resides in the fourth and seventh words. (The extension bit is in the DNTTPOINTER to avoid wasting space due to structure packing rules.)

The second field in each structure is "kind", which acts like a Pascal variant tag to denote the type of the structure. The "unused" fields are just included for clarity.

Followings are different classes of DNTT entries. The whole union "dnttentry" is declared at the end of this section.

7.3.1 File-class ("File") DNTT Entries

- **DNTT_SRCFILE structure definition:**

```
struct DNTT_SRCFILE {          /* 3 words */
    BITS          extension: 1;
    KINDTYPE      kind: 10;
    LANGTYPE      language: 4;
    BITS          unused: 17;
    VTPOINTER     name;
    SLTPOINTER    address;
};
```

Fields definition:

<i>extension:</i>	Always zero.
<i>kind:</i>	always K_SRCFILE type.
<i>language:</i>	Language type.
<i>unused:</i>	17 bits filler to the end of 1st word.
<i>name:</i>	Source/listing file name.
<i>address:</i>	Code and text locations. "address" points to a special SLT entry (for the line number only), but the code location is known from context in the SLT.

One SRCFILE is emitted for the start of each source file, the start of each included file, and the return from each included file. Additional SRCFILE entries must also be output before each DNTT_FUNC entry. This guarantees the debuggers know which file a function came from. Specifically, the definitions and rules are as follows:

Definitions

Source block: Contiguous block of one or more lines of text in a source-file, bounded by beginning or end-of-file or include directives (conceptually identical to the "basic block" in optimizer term). No distinction is made between blocks that contain compilable code and those that don't.

Code segment: Contiguous DNTT block of DNTT (and associated SLT) entries that are generated from the same "source block". "SLT_SRC" is used here to actually refer to an SLT_SPEC entry of type SLT_SRCFILE. Same goes for SLT_FUNC.

Rules

4. One DNTT_SRCFILE and SLT_SRC must be emitted at the head of each code segment to facilitate reading backwards through the DNTT or SLT tables from any point in the segment to determine the enclosing source file. If the source-file changes within the body of a function/subprogram, a DNTT_SRCFILE/SLT_SRC pair must be emitted prior to any additional DNTT or SLT entries generated by the remainder of that function/subprogram.
5. One DNTT_SRCFILE/SLT_SRC pair is always emitted **immediately** before any DNTT_FUNC/SLT_FUNC. Exception: a DNTT_SA and associated DNTT_XREF may appear between a DNTT_FUNC and its preceding DNTT_SRCFILE. There can be nothing between the SLT_SRC and the SLT_FUNC. The DNTT_SRCFILE (preceding the DNTT_FUNC) must name the file containing the function's declaration. The SLT_FUNC must contain the line number of the line in the function's declaration where the function's name appears. This line number must match the line number that appears in the XT record denoting the function's declaration. The SLT_END associated with the SLT_FUNC must contain the line number of the source line containing the scope-closing token (i.e. "}" or "end").
6. One DNTT_SRCFILE/SLT_SRC pair must be emitted for a source file that otherwise would not be mentioned in the DNTT i.e. source files that do not generate a code segment. This is required for static analysis only.

Notes:

- Listing files and listing file line numbers may be used in place of source files and source file line numbers. A special compiler option will designate which is generated by the compiler
- SRCFILE names are exactly as seen by the compiler, i.e. they may be relative, absolute, or whatever. C include file names must be given as absolute paths if found "in the usual place", i.e., /usr/include/...

7.3.2 Code-class ("Scoping") DNTT Entries

- **DNTT_MODULE structure definition:**

```
struct DNTT_MODULE {           /*5 words */
```

```
        BITS           extension: 1;
        KINDTYPE       kind: 10;
        BITS           unused: 21;
        VTPOINTER      name;
        VTPOINTER      alias;
        DNTTPOINTER    dummy;
        SLTPOINTER     address;
};
```

Fields definition:

```
extension:    Always zero.
kind:        always K_MODULE type.
unused:     21 bits filler to the end of 1st word.
name:       Module name.
alias:      Alternate name, if any.
dummy:      4th word must be DNTTPOINTER.
address:    Code and text location.
```

One MODULE is emitted for the start of each Pascal/Modcal module or C source file (C sources are considered a nameless module). "address" points to a special SLT entry, but the code location is known from context in the SLT.

In the case of languages that do not support modules (such as FORTRAN) a DNTT_MODULE and DNTT_END pair are not used. Every MODULE must have a matching END (see below). If a Pascal/Modcal module has a module body (some code), the latter must be represented by a FUNCTION-END pair as well (see below).

For items within a module, the public bit is true if that item is exported by the module. If the public bit of an item is set, that item is visible within any module or procedure that imports the module containing the item. If the public bit of an item is not set, then the item is only visible within the module.

The "dummy" field exists only because the first word of each extension block must be a DNTTPOINTER; it is important only that the extension bit of the DNTTPOINTER be set.

The MODULE DNTT should be used only in the LNTT.

- **DNTT_FUNC structure definition:**

```
struct DNTT_FUNC {
    /*0*/ BITS           extension:1;           /* always zero */
        KINDTYPE       kind: 10;              /* K_FUNCTION, K_ENTRY,
                                                K_BLOCKDATA, or,
```

Symbolic Debug Information

```

                                KMEMFUNC */
        BITS      public: 1;      /* 1 => globally visible */
        LANGTYPE   language: 4;    /* type of language */
        BITS      level: 5;        /* nesting level (top level =0)*/
        BITS      optimize: 2;     /* level of optimization */
        BITS      varargs: 1;      /* ellipses. Pascal/800 later */
        BITS      info: 4;         /* lang-specific stuff; F_xxxx*/
#ifdef CPLUSPLUS
        BITS      inlined: 1;
        BITS      localloc: 1;     /* 0 at top, 1 at end of block */
#endif
#ifdef TEMPLATES
        BITS      expansion: 1;    /* 1 = function expansion */
        BITS      unused: 1;
#else /* TEMPLATES */
        BITS      unused: 2;
#endif /* TEMPLATES */
#else
        BITS      unused: 4;
#endif

/*1*/ VTPOINTER   name;           /* name of function */
/*2*/ VTPOINTER   alias;          /* alternate name, if any */
/*3*/ DNTTPOINTER firstparam;     /* first FPARAM, if any */
/*4*/ SLTPOINTER  address;        /* code and text locations */
/*5*/ ADDRESS     entryaddr;      /* address of entry point */
/*6*/ DNTTPOINTER retval;        /* return type, if any */
/*7*/ ADDRESS     lowaddr;        /* lowest address of function */
/*8*/ ADDRESS     hiaddr;         /* highest address of function */
};                                /* nine words */
```

Struct DNTT_FUNC is used for dfunc and dentry, and dblockdata types.

One FUNCTION or ENTRY is emitted for each formal function declaration (with a body) or secondary entry point, respectively. They are not emitted for bodyless declarations (FORWARD, EXTERNAL, "int x ();" etc.).

A dblockdata is emitted for Fortran BLOCK DATA constructs only. "address" always points to a special SLT entry.

For FUNCTION types, the "entryaddr" field is the code address of the primary entry point of the function. The "lowaddr" field is the lowest code address of the function. The "hiaddr" field is the highest code address of the function. This both gives the size of the function and helps in mapping code locations to functions when there are any-

mous (non-debuggable) functions present. These three fields should be filled in by the generation of fixups.

For ENTRY types, the "entryaddr" field points to the proper code location for calling the function at the secondary entrypoint, and the "lowaddr" and "hiaddr" fields are nil (zero). For a FORTRAN subroutine with alternate entries, DNTT_DVARs are required to represent the parameters, see the DNTT_FPARAM definition for the details.

For BLOCKDATA types, the "public" bit should be set to 1, the "optimize" field should be set to the optimized level when compiling with -O, the "level", "varargs" and "info" fields should all be 0. The "firstparam" field should be DNTTNIL. The "entryaddr" and "lowaddr" fields should be 0, and the "highaddr" field should be FFFFFFFC (-4). The "retval" field should be set to T_UNDEFINED, with length 0. An SLT_FUNCTION/SNT_END pair should be emitted for each DNTT_FUNC (BLOCKDATA).

Every FUNCTION or BLOCKDATA must have a matching END (see below).

For languages in which a functions return value is set by assigning the value to the function name (such as FORTRAN & Pascal), a DVAR entry should also be emitted for the function. The address of this DVAR for the function should be the address of the answer spot for the function. This will allow the user to display the current return value while the function is executing.

The "varargs" field indicates whether the function was declared as having a variable-length parameter list. This is currently possible only via ANSI/C function-prototype "ellipses" (...). The "info" field provides additional language-specific characteristics of the function and/or its parameter-list.

The localloc (local variables location) is currently only used in the following context: If the function language is LANG_CPLUSPLUS, then 0 means that locals are at the beginning of the block, and 1 means that locals appears at the end of a block. For all other languages this bit is not used.

The FUNCTION DNTT should be used only in the LNTT.

- **DNTT_BEGIN structure definition:**

```
struct DNTT_BEGIN {
/*0*/  BITS          extension: 1;          /* always zero */
        KINDTYPE     kind: 10; /           * always K_BEGIN */
#ifdef CPLUSPLUS
        BITS          classflag: 1;         /* beginning of class def'n */
        BITS          unused: 20;
#else
        BITS          unused: 21;
#endif
}
```

Symbolic Debug Information

```
/*1*/ SLTPOINTER    address;          /* code and text locations */
};                                     /* two words */
```

BEGINs are emitted as required to open a new (nested) scope for any type of variable or label, at any level within MODULE-END and FUNCTION-END pairs. Every BEGIN must have a matching END (see below). "address" points to a special SLT entry, but the code location is known from context in the SLT. Because a DNTT BEGIN-END is used to indicate a new scope, the Pascal BEGIN-END pair does not produce a DNTT BEGIN-END, while the C { } construct does.

The BEGIN DNTT should be used only in the LNTT.

- **DNTT_COMMON structure definition:**

```
struct DNTT_COMMON {
/*0*/  BITS          extension: 1;    /* always zero */
      KINDTYPE      kind: 10;       /* always K_COMMON */
      BITS          unused: 21;
/*1*/  VTPOINTER     name;           /* name of common block */
/*2*/  VTPOINTER     alias;          /* alternate name, if any */
};                                     /* three words */
```

COMMONs are used to indicate that a group of variables are members of a given FORTRAN common block. For each common block, a DNTT_COMMON is emitted, followed by a DNTT_SVAR for each member of the common block, and finally a DNTT_END. If type information is required for a member of the common block (such as an array), it may also be within the DNTT_COMMON, DNTT_END pair.

The COMMON DNTT should be used only in the LNTT.

- **DNTT_WITH structure definition:**

```
struct DNTT_WITH {
/*0*/  BITS          extension: 1;    /* always zero */
      KINDTYPE      kind: 10;       /* always K_WITH */
      BITS          addrtype: 2;     /* 0 => STATTYPE */
                                   /* 1 => DYNTYPE */
                                   /* 2 => REGTYPE */
      BITS          indirect: 1;     /* 1 => pointer to object */
      BITS          longaddr: 1;     /* 1 => in long pointer space */
      BITS          nestlevel: 6;    /* # of nesting levels back */
      BITS          unused: 11;
};
```

```
/*1*/ long          location;      /* where stored (allocated) */
/*2*/ SLTPINTER     address;
/*3*/ DNTTPINTER    type;          /* type of with expression */
/*4*/ VTTPINTER     name;          /* name of with expression */
/*5*/ unsigned long offset;        /* byte offset from location */
};                                  /* six words */
```

WITHs are emitted to open a with scope. Like a BEGIN, a WITH requires a matching END to close the scope. A single WITH statement possessing more than one record expression, should be handled as multiple nested withs with only one expression each.

The "addrtype" field indicates the addressing mode used for the record expression, and along with the "indirect" field, tells how to interpret the "location" and "offset" fields. Thus, depending upon the value of "addrtype", "location" may contain a short pointer, an offset from the local frame pointer, or a register number. If "nestlevel" is non-zero and "addrtype" is DYNTYPE, the address for the record expression is computed by tracing back "nestlevel" static links and using "location" as an offset from the frame pointer at that level. (This situation occurs only on the HP9000 FOCUS architecture.) The use of the "offset" field is the same as for the DNTT_SVAR entry (see below). The "type" field is the type of the record expression. The "name" field is the symbolic representation of the record expression (ex. "p[i]^"). "address" points to a special SLT, but the code location is known from context in the SLT.

The WITH DNTT should be used only in the LNTT.

- **DNTT_END structure definition:**

```
struct DNTT_END {
/*0*/  BITS          extension: 1;    /* always zero */
      KINDTYPE      kind: 10;        /* always K_END */
      KINDTYPE      endkind: 10;     /* DNTT kind closing scope for */
#ifdef CPLUSPLUS
      BITS          classflag: 1;    /* end of class def'n */
      BITS          unused: 10;
#else
      BITS          unused: 11;
#endif
/*1*/  SLTPINTER     address;          /* code and text locations */
/*2*/  DNTTPINTER    beginscope;      /* start of scope */
};                                     /* three words */
```

ENDs are emitted as required to close a scope started by a MODULE, FUNCTION, WITH, COMMON, or BEGIN (but not an ENTRY).

Each points back to the DNTT entry that opened the scope. "endkind" indicates which kind of DNTT entry is associated with the END and is filled in by the preprocessor. "address" points to a special SLT entry, but the code location is known from context in the SLT.

The END DNTT should be used only in the LNTT.

- **DNTT_IMPORT structure definition:**

```
struct DNTT_IMPORT {
    /*0*/ BITS      extension: 1;    /* always zero */
        KINDTYPE   kind: 10;      /* always K_IMPORT */
        BITS       explicit: 1;    /* module directly imported */
        BITS       unused: 20;
    /*1*/ VTPOINTER module;         /* module imported from */
    /*2*/ VTPOINTER item;           /* name of item imported */
};                                  /* three words */
```

Within a module, there is one IMPORT entry for each imported module, function, or variable. The item field is nil when an entire module is imported. Used only by Pascal/Modcal. Note that exported functions and variables have their public bits set.

The "explicit" flag indicates the module was directly imported. When not set, the module was imported by an imported module.

The IMPORT DNTT should be used only in the LNTT.

- **DNTT_LABEL structure definition:**

```
struct DNTT_LABEL {
    /*0*/ BITS      extension: 1;    /* always zero */
        KINDTYPE   kind: 10;      /* always K_LABEL */
        BITS       unused: 21;
    /*1*/ VTPOINTER name;           /* name of label */
    /*2*/ SLTPOINTER address;       /* code and text locations */
};                                  /* three words */
```

One LABEL is emitted for each source program statement label, referencing the matching physical line (SLT entry). An SLT pointer is used, instead of just a line-

number, so a code location is known for setting a breakpoint. This is the only case of SLTPONTER that points to a normal (not special) SLT entry.

If a label appears at the very end of a function (after all executable code), a normal SLT entry must be emitted for it anyway. In this case the SLT entry points to an exit (return) instruction.

Numeric labels are named as the equivalent character string with no leading zeroes, except in those languages where the leading zeroes are significant (i.e. COBOL).

The LABEL DNTT should be used only in the LNTT.

7.3.3 Storage-class ("Name") DNTT Entries

- **DNTT_FPARAM structure definition:**

```
struct DNTT_FPARAM {
    /*0*/ BITS      extension:    1;      /* always zero */
           KINDTYPE  kind:        10;     /* always K_FPARAM */
           BITS      regparam:    1;      /* 1 => REGTYPE, */
                                           /* not DYNTYPE */
           BITS      indirect:    1;      /* 1 => pass by reference */
           BITS      longaddr:    1;      /* 1 => in long pointer space */
           BITS      ccopyparam:  1;      /* 1 => Copied to a local */
                                           /* only for fortran strings */

#ifdef CPLUSPLUS
           BITS      dflt:        1;      /* default parameter value? */
           BITS      unused:      16;
#else
           BITS      unused:      17;
#endif
    /*1*/ VTPONTER   name;              /* name of parameter */
    /*2*/ DYNTYPE     location;          /* where stored */
    /*3*/ DNTTPOINTER type;              /* type information */
    /*4*/ DNTTPOINTER nextparam;         /* next FPARAM, if any */
    /*5*/ int         misc;              /* assorted uses */
};                                       /* six words */
```

FPARAMs are chained together in parameter list order (left to right) from every FUNCTION, ENTRY, or FUNCTYPE (see below), one for each parameter, whether or not the type is explicitly declared.

For unnamed parameters, the FPARAM name is "*". "regparam" implies that the storage location given is to be interpreted as a REGTYPE, not a DYNTYPE, that is, the parameter was passed in a register. "indirect" implies that the storage location given contains a data pointer to the parameter described, not the parameter itself, due to a call by reference (Pascal VAR, for instance). In the case where a call-by-value parameter is too big to be passed in the parameter list (e.g., a copied-value parameter in Pascal), the "location" must be given as the actual (post-copy) location of the parameter. "longaddr" is meaningful only for varparams, and indicates that the storage location given contains a 64 bit Spectrum long pointer. The long pointer could be in 2 consecutive words, or in the case of a regparam, two consecutive registers. "copyparam" implies that the parameter has been copied to a local, and thus the location is relative to the sp of the current procedure, not the sp of the previous procedure. "misc" is for assorted values. Currently, if the parameter is of type T_FTN_STRING_S300 then the "misc" field contains the SP relative offset of the word containing the length of the string

In the case of a FORTRAN routine with alternate entries, DNTT DVARs also must be emitted for each parameter. The reason is that with FORTRAN alternate entries, the same parameter can be in two different entry's parameter lists, in a different location (ex. the parameter "x" in "subroutine a(x,y,z)" and "entry b(v,w,x)") and yet they both represent the same parameter. Thus in order to insure a consistent address for such parameters, the compiler allocates a local temporary, and the prologue code for each entry copies the parameters into the local temps. So, to insure that the debugger can find the parameters, a DNTT DVAR must be generated for each temporary, with the name of the DVAR being the name of the FPARAM for which the temp. was allocated.

The FPARAM DNTT should be used only in the LNTT.

- **DNTT SVAR and DVAR structures definition:**

```
struct DNTT_SVAR {
/*0*/  BITS      extension:1;      /* always zero */
        KINDTYPE  kind:10;         /* always K_SVAR */
        BITS      public:1;        /* 1 => globally visible */
        BITS      indirect:1;      /* 1 => pointer to object */
        BITS      longaddr:1;      /* 1 => in long pointer space */
#ifdef CPLUSPLUS
        BITS      staticmem:1;     /* 1 => member of a class */
        BITS      a_union:1;       /* 1 => anonymous union member */
        BITS      unused:16;
#else
        BITS      unused:18;
#endif
/*1*/  VTPOINTER  name;             /* name of object (variable) */
/*2*/  STATTYPE   location;         /* where stored (allocated) */
}
```

Symbolic Debug Information

```
/*3*/ DNTTPOINTER type;          /* type information */
/*4*/ unsigned long   offset;      /* post indirection byte offset */
/*5*/ unsigned long   displacement; /* pre indirection byte offset */
/* six words */
};

struct DNTT_DVAR {
/*0*/ BITS      extension: 1;      /* always zero */
      KINDTYPE  kind:10;          /* always K_DVAR */
      BITS      public:1;         /* 1 => globally visible */
      BITS      indirect:1;       /* 1 => pointer to object */
      BITS      regvar:1;         /* 1 => REGTYPE, not DYNTYPE */
#ifdef CPLUSPLUS
      BITS      a_union:1;        /* 1 => anonymous union member */
      BITS      unused:17;
#else
      BITS      unused:18;
#endif
/*1*/ VTPOINTER  name;            /* name of object (variable) */
/*2*/ DYNTYPE    location;        /* where stored (allocated) */
/*3*/ DNTTPOINTER type;          /* type information */
/*4*/ unsigned long   offset;      /* post indirection byte offset */
/* for use in cobol structures */
/* five words */
};
```

SVARs describe static variables (with respect to storage, not visibility) and DVARs describe dynamic variables, and also describe register variables. Note that SVARs have an extra word, "offset", not needed for the other types. This provides for direct data which is indexed from a base, and indirect data which is accessed through a pointer, then indexed.

The "location" field of an SVAR will require a fixup. An example of when the offset field can be useful, is a FORTRAN common block. In a common block declaration such as "common /marx/ groucho, harpo, chico", the symbol "marx" is the only global symbol. If "marx" is accessed indirectly, then the address of "harpo" would contain the address of "marx" in the location field (with the indirect bit on), and the offset of "harpo" from "marx" in the offset field. If "marx" is not indirect, then location field can be filled in by a fixup of the form address(marx) + offset of harpo, and the offset field is not needed.

The compilers must emit SVARs even for data objects the linker does not know about by name, such as variables in common blocks.

As in the FPARAM entry, the longaddr field indicates the use of a Spectrum long pointer, and is valid only if the indirect flag is true. The "regvar" field also has the same meaning as in the FPARAM case.

For languages in which a functions return value is set by assigning the value to the function name (such as FORTRAN & Pascal), a DVAR entry should also be emitted for the function. The address of this DVAR for the function should be the address of the answer spot for the function. This will allow the user to display the current return value while the function is executing.

For a FORTRAN subroutine with alternate entries, DNTT_DVARs are required to represent the parameters, see the DNTT_FPARAM definition for the details.

The SVAR can be used in both the GNTT and LNTT, while the DVAR is only applicable to the LNTT.

- **DNTT_CONST structure definition:**

```
struct DNTT_CONST {
/*0*/  BITS          extension:1;      /* always zero */
        KINDTYPE     kind:10;         /* always K_CONST */
        BITS         public:1;        /* 1 => globally visible */
        BITS         indirect:1;      /* 1 => pointer to object */
        LOCDESCTYPE  locdesc:3;       /* meaning of location field */
#ifdef CPLUSPLUS
        BITS         classmem:1;      /* 1 => member of a class */
        BITS         unused:15;
#else
        BITS         unused:16;
#endif
/*1*/  VTPOINTER     name;             /* name of object */
/*2*/  STATTYPE      location;         /* where stored */
/*3*/  DNTTPOINTER   type;            /* type information */
/*4*/  unsigned long  offset;          /* post indirection byte offset */
/*5*/  unsigned long  displacement;    /* pre indirection byte offset */
};                                     /* six words */
```

The value of locdesc determines the meaning of location. Compilers are free to use any of the three types (LOC_IMMED, LOC_PTR, LOC_VT) as feasible and appropriate. They might, for example, merely dump all CONST values into the VT, with

some redundancy, if they could do no better. Ideally, each compiler would use all three types according to whether the constant is stored in an immediate instruction (so a copy is needed here), in code or data space, or nowhere else, respectively.

If `locdesc == LOC_PTR`, `CONST` is very much like an `SVAR`, and the indirect and offset values are relevant.

The `CONST DNTT` can be used in both the `GNTT` and `LNTT`.

7.3.4 Type-class ("Type") DNTT Entries

- **DNTT_TYPE structure definition:**

```
struct DNTT_TYPE {
/*0*/  BITS          extension:1;    /* always zero */
      KINDTYPE      kind:10;        /* either K_TYPEDEF or */
                                           /* K_TAGDEF */
      BITS          public:1;       /* 1 => globally visible */
      BITS          typeinfo:1;     /* 1 => type info available */
      BITS          unused:19;
/*1*/  VTPOINTER     name;          /* name of type or tag */
/*2*/  DNTTPOINTER  type;          /* type information */
};                                   /* three words */
```

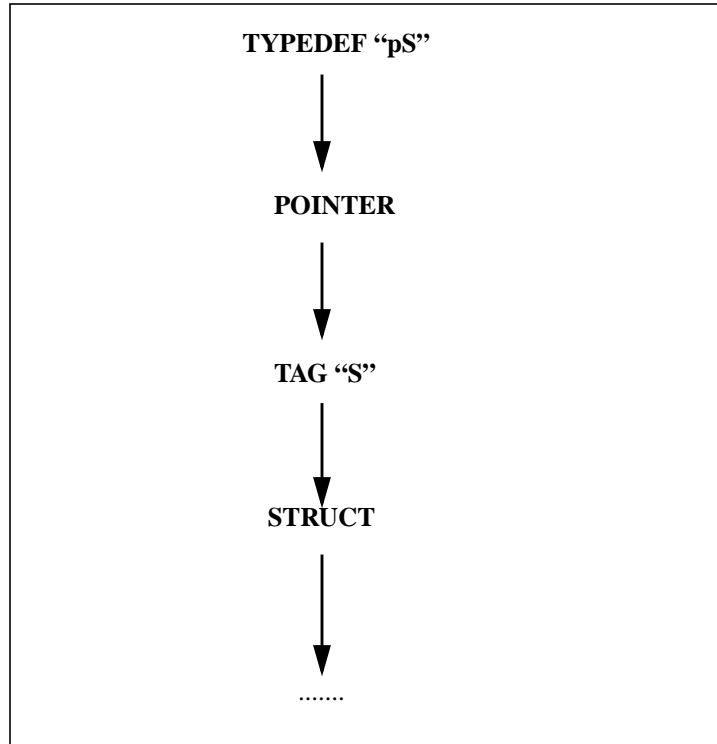
The `DNTT_TYPE` type is used for `dtype` and `dtag` entries. `TYPEDEFs` are just a way of remembering names associated with types declared in Pascal, via "type" sections, or in C, via "typedef"s. `TAGDEFs` are used for C "struct", "union", and "enum" tags, which may be named identically to "typedef"s in the same scope. `TAGDEFs` always point at `STRUCTs`, `UNIONs`, or `ENUMs` (see below), and provide a way to "hang" a name onto a subtree.

Note that named types point directly to the underlying structures, not to intervening `TYPEDEFs` or `TAGDEFs`. Type information in `TYPEDEFs` and `TAGDEFs` point to the same structures independent of named instantiations of the types.

For example:

```
typedef struct S {...} *pS;
```

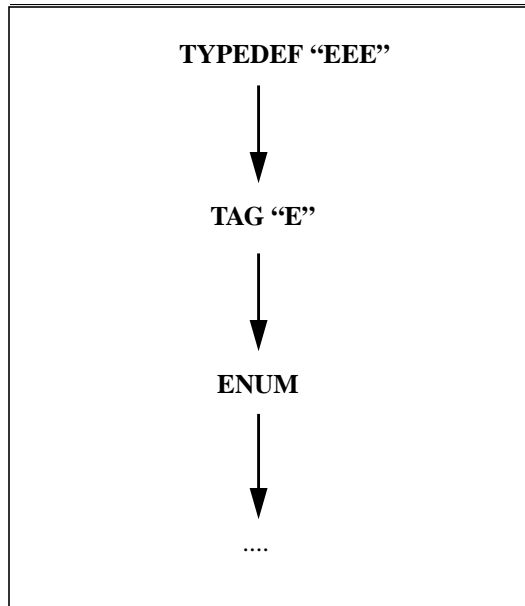
would generate something like this:



And:

```
typedef enum E { ... } EEE;
```

would generate something like this:



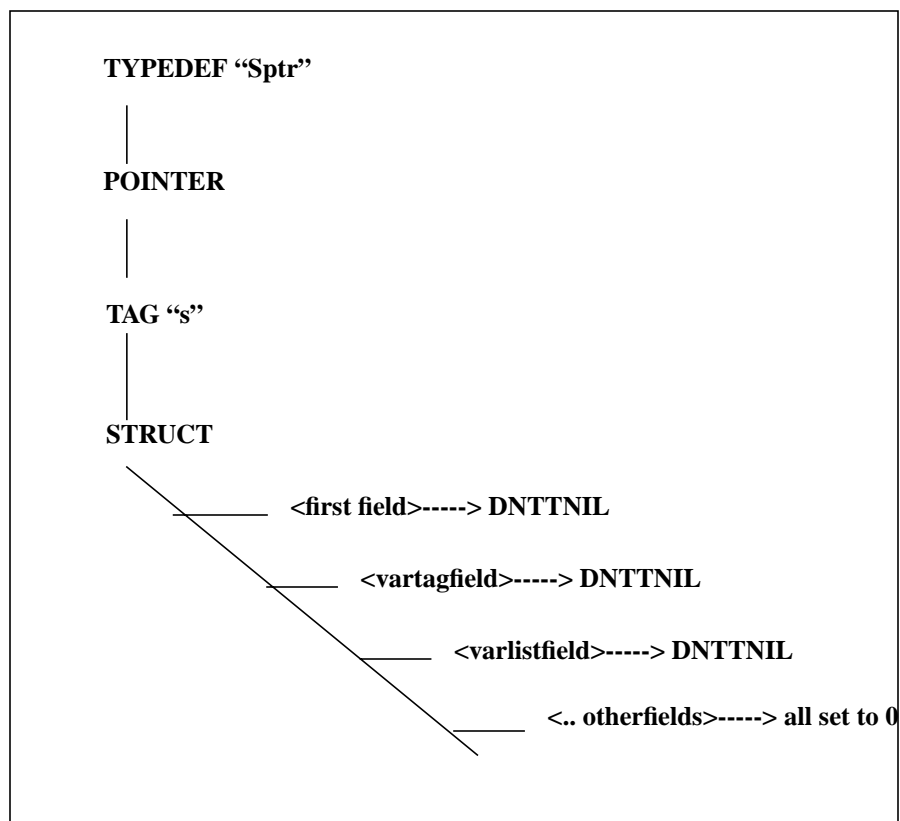
Note also that variables (of a named non-base type) must point to TYPEDEF or TAG-DEF dntt, and not the underlying structures. If this is not done, the removal of duplicate global information is impossible.

The "typeinfo" flag only applies to TAGDEFs. When not set, it is used to indicate that an underlying struct, union, or enum is named, but the actual type is not declared. In general, "typeinfo" will be set to 1. It will be set to a 0 if the type subtree is not available. Consider the C file

```
*typedef struct s *Sptr;  
  
*main(){ }
```

which is a valid compilation unit with "struct s" defined in another file. For this case, the "typeinfo" for TAGDEF "s" will be set to 0, and "type" points to a "nil" DNTT_STRUCT (i.e. a DNTT_STRUCT entry with its "firstfield", "vartagfield", and "varlist" fields set to DNTTNIL and its "declaration" and "bitlength" fields set to 0).

Graphically:



Thus, whenever "typeinfo" is 0, "type" must point to an appropriate DNTT entry which has all its fields correctly NIL'ed. This applies to *named* DNTT_STRUCT's, DNTT_UNION's, and DNTT_ENUM's.

The TYPEDEF and TAGDEF DNTTs may be used in both the GNTT and LNTT.

- **DNTT_POINTER structure definition:**

```
struct DNTT_POINTER {
/*0*/  BITS          extension: 1;    /* always zero */
#ifdef CPLUSPLUS
    KINDTYPE        kind: 10;        /* K_POINTER or K_REFERENCE */
#else
    KINDTYPE        kind: 10;        /* always K_POINTER */
#endif
}
```

```
#endif
        BITS            unused: 21;
/*1*/ DNTTPOINTER pointsto;    /* type of object */
/*2*/ unsigned long    bitlength;    /* size of pointer, not object */
};                                /* three words */
```

- **DNTT ENUM and MEMENUM structures definition:**

```
struct DNTT_ENUM {
/*0*/  BITS            extension: 1;    /* always zero */
        KINDTYPE      kind: 10;    /* always K_ENUM */
        BITS            unused: 21;
/*1*/  DNTTPOINTER firstmem;    /* first MEMENUM (member) */
/*2*/  unsigned long    bitlength;    /* packed size */
};                                /* three words */
```

```
struct DNTT_MEMENUM {
/*0*/  BITS            extension: 1;    /* always zero */
        KINDTYPE      kind: 10;    /* always K_MEMENUM */
#ifdef CPLUSPLUS
        BITS            classmem: 1;    /* 1 => member of a class */
        BITS            unused: 20;
#else
        BITS            unused: 21;
#endif
/*1*/  VTPOINTER      name;    /* name of member */
/*2*/  unsigned long    value;    /* equivalent number */
/*3*/  DNTTPOINTER nextmem;    /* next MEMENUM, else */
                                   /* ENUM type */
};                                /* four words */
```

Each ENUM begins a chain of (name, value) pairs. The nextmem field of the last memenum, should be DNTT NIL. The POINTER, ENUM, and MEMENUM DNTTs can all be used in both the GNTT and LNTT.

- **DNTT SET, SUBRANGE, and ARRAY structures definition:**

```
struct DNTT_SET {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE     kind:10;       /* always K_SET */
        BITS         declaration:2; /* normal, packed, or crunched */
        BITS         unused:19;
/*1*/  DNTTPOINTER subtype;         /* type implies bounds of set */
/*2*/  unsigned long  bitlength;     /* packed size */
};                                     /* three words */

struct DNTT_SUBRANGE {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE     kind:10;       /* always K_SUBRANGE */
        BITS         dyn_low:2;     /* >0 => nonconstant low bound */
        BITS         dyn_high:2;    /* >0 => nonconstant high bound */
        BITS         unused: 17;
/*1*/  long          lowbound;       /* meaning depends on subtype */
/*2*/  long          highbound;      /* meaning depends on subtype */
/*3*/  DNTTPOINTER subtype;         /* immediate type or ENUM */
/*4*/  unsigned long  bitlength;     /* packed size */
};                                     /* five words */

struct DNTT_ARRAY {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE     kind: 10;      /* always K_ARRAY */
        BITS         declaration: 2; /* normal, packed, or crunched */
        BITS         dyn_low: 2;    /* >0 => nonconstant low bound */
        BITS         dyn_high: 2;   /* >0 => nonconstant high bound */
        BITS         arrayisbytes:1; /* 1 => array size is in bytes */
        BITS         elemisbytes: 1; /* 1 => elem. size is in bytes */
        BITS         elemorder: 1;  /* 0 => in increasing order */
        BITS         justified: 1;   /* 0 => left justified */
        BITS         unused: 11;
/*1*/  unsigned long  arraylength;    /* size of whole array */
/*2*/  DNTTPOINTER indextype;        /* how to index the array */
};
```

```
/*3*/ DNTTPOINTER elemtype;      /* type of each array element */
/*4*/ unsigned long  elemlength;  /* size of one element */
};                                /* five words */
```

The `dyn_low` and `dyn_high` fields are non-zero only if the `DNTT_SUBRANGE` is defining the range of an array index, otherwise they are always zero. The `dyn_low` and `dyn_high` bits are duplicated in the `DNTT_SUBRANGE` defining the range of the array index (so `sllic` can fix the pointers). "dyn_low" indicates whether the lower bound for the subscript of the array is dynamic. If the `dyn_low` field is zero, then the `lowbound` field of the `DNTT_SUBRANGE` entry, pointed to by the `indextype` field in the `DNTT_ARRAY` entry, is interpreted as a constant lower bound. If the `dyn_low` field is 1, then the `lowbound` field of the `DNTT_SUBRANGE` is interpreted as a `DYNTYPE` giving a local address where the lower bound can be found. If the `dyn_low` field is 2, then the `lowbound` field of the `DNTT_SUBRANGE` is interpreted as a `DNTTPOINTER` to a variable whose value is the lower bound (needed if the lower bound is a static variable). The `dyn_low` value of 3 is not used. The "dyn_high" bit has a similar meaning relating to the upper bound. If an upper bound for an array parameter is not given (like assumed size arrays in FORTRAN, or "char foo[]" in C) then the upper bound in the `DNTT_SUBRANGE` should be the largest integer that fits in a long integer, so that any value the user can give is legal.

"arrayisbytes" indicates that the field "arraylength" contains the length in bytes rather than bits. This is needed on Spectrum where an array could be up to 2^{32} bytes. A value of zero for `bitsize` will be used to represent 2^{32} .

"elemisbytes" indicates that the field "elemlength" contains the elem. length in bytes rather than bits. The "elemlength" field contains the not the "true" size of an array element, but the size allocated to each element within the array (the "true" size plus any wasted bits on the left or right). As an example for a

Pascal array of a 13 bit structure, the array element size might equal 16, with the justified field equal to 0 to indicate the structure is left justified within the 16 bits. The "true" size of the structure would be found in the `size` field of the

`DNTT_STRUCT` pointed to by the "elemtype" field of the `DNTT_ARRAY`.

"indextype" typically points to a `SUBRANGE` for bounds.

"elemtype" may point to another `ARRAY` for multi-dimensional arrays. Row or column precedence in the language is reflected in the order of the `ARRAY` entries on the chain. For example, in Pascal, which is row-precedent, an array declared [1..2, 3..4, 5..6] would result in "array 1..2 of array 3..4 of array 5..6 of ...". The same declaration in FORTRAN, which is column-precedent, would result in "array 5..6 of array 3..4 of array 1..2 of ...". This makes index-to-address conversion much easier. Either way an expression handler must know the precedence for the language.

The `SET`, `SUBRANGE`, and `ARRAY` `DNTTs` can be used in both the `GNTT` and `LNTT`.

- **DNTT STRUCT structure definition:**

```
struct DNTT_STRUCT {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE     kind:10;       /* always K_STRUCT */
        BITS          declaration:2; /* normal, packed, or crunched */
        BITS          unused:19;
/*1*/  DNTTPOINTER firstfield;      /* first FIELD, if any */
/*2*/  DNTTPOINTER vartagfield;     /* variant tag FIELD, or type */
/*3*/  DNTTPOINTER varlist;         /* first VARIANT, if any */
/*4*/  unsigned long  bitlength;     /* total at this level */
};                                   /* five words */
```

The "declaration", "vartagfield", and "varlist" fields apply to Pascal/Modcal records only and are nil for record structures in other languages. If there is a tag, then the "vartagfield" points to the FIELD DNTT describing the tag. Otherwise, the "vartagfield" points to the tag type.

The STRUCT DNTT may be used in both the GNTT and LNTT.

- **DNTT UNION structure definition:**

```
struct DNTT_UNION {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE     kind:10;       /* always K_UNION */
        BITS          unused: 21;
/*1*/  DNTTPOINTER firstfield;      /* first FIELD entry */
/*2*/  unsigned long  bitlength;     /* total at this level */
};                                   /* three words */
```

This type of DNTT_UNION supports C unions only and is not used otherwise.

Since STRUCTURES and UNIONS are not packable inside of outer STRUCTURES and UNIONS, their bitlengths tell their actual (not necessarily packed) size, according only as to how they are internally packed.

The UNION DNTT may be used in both the GNTT and LNTT.

- **DNTT FIELD structure definition:**

```
struct DNTT_FIELD {
/*0*/  BITS          extension: 1;    /* always zero */
        KINDTYPE     kind: 10;       /* always K_FIELD */
#ifdef CPLUSPLUS
        BITS          visibility:2;    /* pub = 0, prot = 1, priv = 2 */
        BITS          a_union:1;      /* 1 => anonymous union member */
#ifdef TEMPLATES

        BITS          staticMem:1;    /* 1 -> static member of a template */
        BITS          unused:17;
        #else /* TEMPLATES */
        BITS          unused:18;
        #endif /* TEMPLATES */

        #else /* normal code, not C++ support */
        BITS          unused:21;
        #endif
/*1*/  VTPOINTER     name;            /* name of field, if any */
/*2*/  unsigned long  bitoffset;      /* of object itself in STRUCT */
/*3*/  DNTTPOINTER    type;           /* type information */
/*4*/  unsigned long  bitlength;      /* size at this level */
/*5*/  DNTTPOINTER    nextfield;      /* next FIELD in STRUCT, if any */
};                                     /* six words */
```

This type describes the fields in Pascal records and C structures and unions. The bitoffset is from the start of the STRUCT or UNION that started the chain, to the start of the object itself, ignoring any padding. Note that bitoffset does not have to be on a byte boundary. For unions, each bitoffset should be zero since all fields overlap.

The bitlength field is the same as that of the type except for C bit fields, which may be a different size than the base type.

The FIELD DNTT can be used in both the GNTT and LNTT.

- **DNTT VARIANT structure definition:**

```
struct DNTT_VARIANT {
/*0*/  BITS          extension: 1;    /* always zero */
        KINDTYPE     kind: 10;       /* always K_VARIANT */
        BITS          unused: 21;
/*1*/  long           lowvarvalue;    /* meaning depends on vartype */
/*2*/  long           hivarvalue;     /* meaning depends on vartype */
/*3*/  DNTTPOINTER   varstruct;      /* this variant STRUCT, if any */
/*4*/  unsigned long  bitoffset;      /* of variant, in outer STRUCT */
/*5*/  DNTTPOINTER   nextvar;        /* next VARIANT, if any */
};                                       /* six words */
```

"varstruct" points to the STRUCT which in turn describes the contents of the variant. The latter might in turn point to VARIANTS of its own, and to FIELDS which point to other STRUCTs.

"lowvarvalue" and "hivarvalue" are the range of values for which this variant applies; more than one dntt VARIANT may be necessary to describe the range (e.g., 'a'..'n','q:'). A type field is unnecessary, as the type can be obtained from the "vartagfield" field of the STRUCT DNTT.

The VARIANT DNTT can be used in both the GNTT and LNTT.

- **DNTT FILE structure definition:**

```
struct DNTT_FILE {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE     kind:10;       /* always K_FILE */
        BITS          ispacked: 1;   /* 1 => file is packed */
        BITS          unused: 20;
/*1*/  unsigned long  bitlength;     /* of whole element buffer */
/*2*/  unsigned long  bitoffset;     /* of current element in buffer */
/*3*/  DNTTPOINTER   elemtype;      /* type and size of of element */
};                                       /* four words */
```

Pascal/Modcal is the only language of interest with built-in file buffering. For Pascal/Modcal files, the symbol table tells the file element type, the sizes of the current element (via "elemtype") and the whole buffer (via "bitlength"), and the locations of the element buffer (from the parent "NAME" entry) and the element itself within the buffer, following header information (from "bitoffset").

The FILE DNTT can be used in both the GNTT and LNTT.

- **DNTT FUNCTTYPE structure definition:**

```
struct DNTT_FUNCTYPE {
/*0*/  BITS          extension:1;    /* always zero */
      KINDTYPE      kind:10;        /* always K_FUNCTYPE */
      BITS          varargs:1;      /* func-proto ellipses. */
      BITS          info:4;         /* lang-specific stuff; F_xxxx */
      BITS          unused:16;
/*1*/  unsigned long  bitlength;     /* size of function pointer */
/*2*/  DNTTPOINTER  firstparam;     /* first FPARAM, if any */
/*3*/  DNTTPOINTER  retval;         /* return type, if any */
};                                     /* four words */
```

This type supports function variables in a limited way, including the parameter types (if any) and the return value type (if any).

See DNTT_FUNC for discussion of various fields.

The FUNCTYPE DNTT can be used in both the GNTT and LNTT.

- **DNTT COBSTRUCT structure definition:**

```
struct DNTT_COBSTRUCT {
/*0*/  BITS          extension:1;    /* always zero */
      KINDTYPE      kind:10;        /* always K_COBSTRUCT */
      BITS          hasoccurs:1;     /* descendant has OCCURS clause */
      BITS          istable: 1;      /* is a table item? */
      BITS          unused:19;
/*1*/  DNTTPOINTER  parent;         /* next higher data item */
/*2*/  DNTTPOINTER  child;          /* 1st descendant data item */
};
```

Symbolic Debug Information

```
/*3*/ DNTTPOINTER sibling;          /* next data item at this level */
/*4*/ DNTTPOINTER synonym;         /* next data item w/ same name */
/*5*/ BITS          catusage: 6;    /* category or usage of item */
      BITS          pointloc:8;     /* location of decimal point */
      BITS          numdigits:10;   /* number of digits */
      BITS          unused2:8;
/*6*/ DNTTPOINTER table;           /* array entry describing table */
/*7*/ VTPOINTER     editpgm;        /* name of edit subprogram */
/*8*/ unsigned long bitlength;      /* size of item in bits */
};                                  /* nine words */
```

This entry is used to describe COBOL data items and table items.

A Cobol variable will begin with a DNTT_SVAR, DNTT_DVAR, or DNTT_FPARAM whose "type" field is a DNTTPOINTER to a DNTT_COBSTRUCT.

"parent", "child", "sibling", and "synonym" are DNTTPOINTER to other DNTT_SVAR, DNTT_DVAR, or DNTT_FPARAMs having these particular relationships with the current DNTT_COBSTRUCT (or are set to DNTTNIL if no such relationship exists).

"hasoccurs" is set to 1 if the descendent of this COBOL element (pointed to by "child") has an OCCURS ... DEPENDING ON clause.

"istable" is set to 1 if this COBOL data item is a table. In this case, "table" will point to a DNTT_ARRAY entry describing the table.

The COBSTRUCT DNTT can be used in both the GNTT and LNTT.

- **DNTT MODIFIER structure definition:**

```
struct DNTT_MODIFIER {
/*0*/  BITS          extension:1;    /* always zero */
      KINDTYPE      kind: 10;       /* always K_MODIFIER */
      BITS          m_const:1;      /* const */
      BITS          m_static:1;     /* static */
      BITS          m_void:1;       /* void */
      BITS          m_volatile: 1;  /* volatile */
      BITS          m_duplicate:1;  /* duplicate */
      BITS          unused:16;
/*1*/  DNTTPOINTER type;            /* subtype */
```

```
}; /* 2 words */
```

- The following DNTTs :

**DNTT_GENFIELD,
DNTT_MEMACCESS,
DNTT_VFUNC,
DNTT_CLASS_SCOPE,
DNTT_FRIEND_CLASS,
DNTT_FRIEND_FUNC,
DNTT_CLASS,
DNTT_TEMPLATE,
DNTT_TEMPL_ARG,
DNTT_PTRMEM,
DNTT_INHERITANCE,
DNTT_OBJECT_ID**

are defined to support C++ and template.

```
struct DNTT_GENFIELD {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE     kind:10;       /* always K_GENFIELD */
        BITS          visibility:2;  /* pub = 0, prot = 1, priv = 2 */
        BITS          a_union:1;    /* 1 => anonymous union member */
        BITS          unused:18;
/*1*/  DNTTPOINTER field;           /* pointer to field or qualifier */
/*2*/  DNTTPOINTER nextfield;       /* pointer to next field */
}; /* three words */
```

```
struct DNTT_MEMACCESS {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE     kind:10; /      * always K_MEMACCESS */
        BITS          unused:21;
/*1*/  DNTTPOINTER classptr;        /* pointer to base class */
/*2*/  DNTTPOINTER field;           /* pointer field */
}; /* three words */
```

```
struct DNTT_VFUNC {
/*0*/  BITS          extension: 1;   /* always zero */
```

Symbolic Debug Information

```
        KINDTYPE      kind:10;      /* always K_VFUNCTION */
        BITS          pure:1;      /* pure virtual function ? */
        BITS          unused:20;
/*1*/ DNTTPOINTER funcptr;      /* function name */
/*2*/ unsigned long  vtbl_offset;  /* offset into vtbl for virtual */
};                               /* three words */

struct DNTT_CLASS_SCOPE {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE      kind:10;      /* always K_CLASS_SCOPE */
        BITS          unused:21;
/*1*/  SLTPOINTER    address;        /* pointer to SLT entry */
/*2*/  DNTTPOINTER    type;          /* pointer to class type DNTT */
};                               /* three words */

struct DNTT_FRIEND_CLASS {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE      kind: 10;      /* always K_FRIEND_CLASS */
        BITS          unused: 21;
/*1*/  DNTTPOINTER    classptr;      /* pointer to class DNTT */
/*2*/  DNTTPOINTER    next;          /* next DNTT_FRIEND */
};                               /* three words */

struct DNTT_FRIEND_FUNC {
/*0*/  BITS          extension:1;    /* always zero */
        KINDTYPE      kind:10;      /* always K_FRIEND_FUNC */
        BITS          unused:21;
/*1*/  DNTTPOINTER    funcptr;      /* pointer to function */
/*2*/  DNTTPOINTER    classptr;      /* pointer to class DNTT */
/*3*/  DNTTPOINTER    next;          /* next DNTT_FRIEND */
};                               /* four words */

struct DNTT_CLASS {
/*0*/  BITS          extension:1;    /* always zero */
```

Symbolic Debug Information

```
        KINDTYPE    kind:10;        /* always K_CLASS */
        BITS        abstract:1;     /* is this an abstract class? */
        BITS        class_decl:2;   /* 0=class,1=union,2=struct */
#ifdef TEMPLATES
        BITS        expansion:1;     /* 1=template expansion */
        BITS        unused:17;
#else /* TEMPLATES */
        BITS        unused:18;
#endif /* TEMPLATES */
/*1*/ DNTTPOINTER memberlist;      /* ptr to chain of K_[GEN]FIELDS */
/*2*/ unsigned long  vtbl_loc;      /* offset in obj of ptr to vtbl */
/*3*/ DNTTPOINTER parentlist;      /* ptr to K_INHERITANCE list */
/*4*/ unsigned long  bitlength;     /* total at this level */
/*5*/ DNTTPOINTER identlist;       /* ptr to chain of class ident's */
/*6*/ DNTTPOINTER friendlist;      /* ptr to K_FRIEND list */
#ifdef TEMPLATES
/*7*/ DNTTPOINTER templateptr;     /* ptr to template */
/*8*/ DNTTPOINTER nextexp;         /* ptr to next expansion */
#else /* TEMPLATES */
/*7*/ unsigned long  future2;
/*8*/ unsigned long  future3;
#endif /* TEMPLATES */
};                                  /* nine words */

struct DNTT_TEMPLATE {
/*0*/ BITS          extension:1;    /* always zero */
        KINDTYPE    kind:10;        /* always K_TEMPLATE */
        BITS        abstract:1;     /* is this an abstract class? */
        BITS        class_decl: 2;  /* 0=class,1=union,2=struct */
        BITS        unused:18;
/*1*/ DNTTPOINTER memberlist;      /* ptr to chain of K_[GEN]FIELDS */
/*2*/ long          unused2;        /* offset in obj of ptr to vtbl */
/*3*/ DNTTPOINTER parentlist;      /* ptr to K_INHERITANCE list */
/*4*/ unsigned long  bitlength;     /* total at this level */
/*5*/ DNTTPOINTER identlist;       /* ptr to chain of class ident's */
/*6*/ DNTTPOINTER friendlist;      /* ptr to K_FRIEND list */
/*7*/ DNTTPOINTER arglist;         /* ptr to argument list */
/*8*/ DNTTPOINTER expansions;      /* ptr to expansion list */
```

Symbolic Debug Information

```
}; /* 9 words */
```

DNTT_TEMPLATES only appear in the GNTT. Functions and classes templates cannot be local. (Their instantiations may be).

```
struct DNTT_TEMPL_ARG {
/*0*/  BITS          extension:1; /* always zero */
      KINDTYPE      kind:10;     /* always K_TEMPL_ARG */
      BITS          usagetype:1; /* 0 type-name 1 expression */
      BITS          unused:20;
/*1*/  VTPOINTER     name;        /* name of argument */
/*2*/  DNTTPOINTER   type;        /* for non type arguments */
/*3*/  DNTTPOINTER   nextarg;     /* Next argument if any */
/*4*/  long          unused2[2];
}; /* 6 words */
```

Pxdb fills in the prevexp, and nextexp in the DNTT_CLASS. Pxdb also fills in the expansions field in the DNTT_TEMPLATE.

```
struct DNTT_PTRMEM {
/*0*/  BITS          extension:1; /* always zero */
      KINDTYPE      kind:10;     /* K_PTRMEM or
                                /* K_PTRMEMFUNC */
      BITS          unused:21;
/*1*/  DNTTPOINTER   pointsto;    /* pointer to class DNTT */
/*2*/  DNTTPOINTER   memtype;     /* type of member */
}; /* three words */
```

```
struct DNTT_INHERITANCE {
/*0*/  BITS          extension:1; /* always zero */
      KINDTYPE      kind:10;     /* K_INHERITANCE */
      BITS          Virtual:1;   /* virtual base class ? */
      BITS          visibility:2; /* pub = 0, prot = 1, priv = 2 */
      BITS          unused:18;
/*1*/  DNTTPOINTER   classname;   /* first parent class, if any */
/*2*/  unsigned long  offset;     /* offset to start of base class */
/*3*/  DNTTPOINTER   next;        /* pointer to next K_INHERITANCE */
}; /* four words */
```

```
struct DNTT_OBJECT_ID {
```

```

/*0*/ BITS          extension:1;    /* always zero */
      KINDTYPE      kind:10;        /* K_OBJECT_ID */
      BITS          unused:21;
/*1*/ unsigned long  object_ident;  /* object identifier */
/*2*/ unsigned long  offset;        /* offset to start of base class */
/*3*/ DNTTPOINTER next;            /* pointer to next K_OBJECT_ID */
/*4*/ unsigned long  segoffset;     /* for linker fixup */
};                                  /* five words */

```

7.3.5 General ("overall") DNTT Entry Format

- **Generic Entry for Easy Access:**

```

struct DNTT_GENERIC {
    unsigned long    word [9];      /* rounded up to
};                                  /* whole number of blocks */

struct DNTT_BLOCK {                /* easy way to deal with one block */
/*0*/ BITS          extension: 1;   /* always zero */
      KINDTYPE      kind:10;        /* kind of dnttentry */
      BITS          unused:21;
/*1*/ unsigned long  word [2];
};

```

- **Overall DNTT entry:**

```

union dnttentry {
    struct  DNTT_SRCFILE      dsfile;
    struct  DNTT_MODULE      dmodule;
    struct  DNTT_FUNC         dfunc;
    struct  DNTT_FUNC         dentry;
    struct  DNTT_FUNC         dblockdata;
    struct  DNTT_BEGIN        dbegin;
    struct  DNTT_END          dend;
    struct  DNTT_IMPORT        dimport;
    struct  DNTT_LABEL        dlabel;
    struct  DNTT_WITH          dwith;
    struct  DNTT_COMMON       dcommon;
}

```

Symbolic Debug Information

```
struct DNTT_FPARAM      dfparam;
struct DNTT_SVAR         dsvar;
struct DNTT_DVAR         ddvar;
struct DNTT_CONST        dconst;
struct DNTT_TYPE         dtype;
struct DNTT_TYPE         dtag;
struct DNTT_POINTER      dptr;
struct DNTT_ENUM          denum;
struct DNTT_MEMENUM       dmember;
struct DNTT_SET           dset;
struct DNTT_SUBRANGE      dsubr;
struct DNTT_ARRAY         darray;
struct DNTT_STRUCT        dstruct;
struct DNTT_UNION          dunion;
struct DNTT_FIELD         dfield;
struct DNTT_VARIANT        dvariant;
struct DNTT_FILE          dfile;
struct DNTT_FUNC_TYPE     dfunc_type;
struct DNTT_COBSTRUCT     dcobstruct;

#ifdef CPLUSPLUS
struct DNTT_CLASS_SCOPE   dclass_scope;
struct DNTT_POINTER       dreference;
struct DNTT_PTRMEM        dptrmem;
struct DNTT_PTRMEM        dptrmemfunc;
struct DNTT_CLASS          dclass;
struct DNTT_GENFIELD       dgenfield;
struct DNTT_VFUNC          dvfunc;
struct DNTT_MEMACCESS      dmemaccess;
struct DNTT_INHERITANCE    dinheritance;
struct DNTT_FRIEND_CLASS   dfriend_class;
struct DNTT_FRIEND_FUNC     dfriend_func;
struct DNTT_MODIFIER        dmodifier;
struct DNTT_OBJECT_ID      dobject_id;
struct DNTT_FUNC           dmemfunc;

#ifdef TEMPLATES
struct DNTT_TEMPLATE        dtemplate;
struct DNTT_TEMPL_ARG       dtempl_arg;
struct DNTT_FUNC_TEMPLATE   dfunc_template;
struct DNTT_LINK            dlink; /* generic */
```

```
        struct    DNTT_TFUNC_LINK    dtflink;
#endif /* TEMPLATES */

#endif /* CPLUSPLUS */

        struct    DNTT_XREF            dxref; /* Static analysis info section */
        struct    DNTT_SA              dsa;  /* Static analysis info section */

        struct    DNTT_GENERIC         dgeneric;
        struct    DNTT_BLOCK           dblock;
};
```

7.4 Static Analysis Information

7.4.1 XREF Table (XT) Entry Format

This table contains static information about each named object in a compilation unit. It consists of a collection of lists, each list associated with a DNTT object via the DNTT_XREF that follows the object. The DNTT_XREF contains an XREF-POINTER which is an offset into the XT table, and denotes the beginning of the reference list.

Each list is actually one or more of linear sub-list that are linked together. Each sub-list begins with an XREFNAME entry, which names a (current) source file. Following the XREFNAME is one or more XREFINFO entries, one for each appearance of the object's name in the current file. These entries list what type of reference and the line no. within the file. Column numbers are currently unsupported. The XREFINFO1 structure is normally used.

The XREFINFO2A/B structure pair is only used for compilers which support line numbers greater than 16 bits long. An XREFLINK marks the end of a sublist, so a typical sequence looks like:

XREFNAME, XREFINFO1, XREFINFO1, ... , XREFLINK

Note that all elements of a sublist must appear in sequence (linearly). If the list must be continued, the XREFLINK serves as a continuation pointer from one sublist to the next, and contains another offset into the XT where the next sublist is found for the same named object. If there is no additional sublist, the XREFLINK contains a 0 index, denoting the end of the current list.

Lists for the same named object may appear in different compilation units. It is the responsibility of PXDB to link these together.

Symbolic Debug Information

```
struct XREFINFO1 {
    BITS    tag: 3;                /* always XINFO1 */
    BITS    definition: 1;         /* True => definition*/
    BITS    declaration:1;         /* True => declaration*/
    BITS    modification:1;        /* True => modification*/
    BITS    use:1;                 /* True => use*/
    BITS    call:1;                /* True => call */
    BITS    column:8;              /* Unsigned Byte for Column
                                   /* within line */
    BITS    line:16;               /* Unsigned 16-bits for line # relative */
                                   /* to beginning of current include file. */
};

struct XREFINFO2A {
    /* first word */
    BITS    tag:3;                 /* always XINFO2A */
    BITS    definition:1;          /* True => definition*/
    BITS    declaration: ;         /* True => declaration*/
    BITS    modification:1;        /* True => modification*/
    BITS    use:1;                 /* True => use */
    BITS    call:1;                /* True => call */
    BITS    extra:16;
    BITS    column:8;
};

struct XREFINFO2B {
    /* second word */
    BITS    line:32;               /* Unsigned 32-bits for line # relative */
                                   /* to beginning of current file.    */
};

struct XREFLINK {
    BITS    tag:3;                 /* always XLINK for XREFLINK */
    BITS    next:29;               /* index of next list. If */
                                   /* zero then this is the end of line. */
                                   /* a.k.a. continuation pointer */
};

struct XREFNAME {
```

```
        BITS    tag:3;           /* always XNAME for XREFNAME */
        BITS    filename:29;     /* VTPOINTER to file name */
};

union xrefentry {
    struct XREFINFO1    xrefshort;
    struct XREFINFO2A   xreflong;
    struct XREFINFO2B   xrefline;
    struct XREFLINK     xlink;
    struct XREFNAME     xfname;
};
```

7.4.2 Static Analysis Support DNTT Entries

Static analysis support consists of two DNTT entries:

- **DNTT XREF Entry:**

```
struct DNTT_XREF {
/*0*/  BITS    extension: 1;     /* always zero */
        KINDTYPE kind:10;       /* always K_XREF */
        BITS    language:4;     /* language of DNTT object */
        BITS    unused: 17;
/*1*/  XREFPOINTER xreflist;    /* index into XREF subspace */
/*2*/  long     extra;          /* free */
};                                     /* three words */
```

This entry is used to retrieve cross-reference information from the XREF Table (XT). A DNTT_XREF entry immediately follows the DNTT_SVAR, DNTT_DVAR, DNTT_TYPE, etc. entry to which it pertains.

The XREFPOINTER points into the XT table where the information about the previous DNTT entry is contained. If no entries are generated in the XT table, the xreflist field should contain XREFNIL. The language field contains the source language (LANG_XXX) value of the DNTT object.

The XREF DNTT can be used in both the GNTT and LNTT.

- **DNTT SA Entry:**

```
struct DNTT_SA {  
  /*0*/  BITS          extension: 1;    /* always zero */  
          KINDTYPE     kind: 10;       /* always K_SA   */  
          KINDTYPE     base_kind:10;   /* K_FUNCTION, K_LABEL, etc */  
          BITS          unused: 11;  
  /*1*/  VTPOINTER     name;  
  /*2*/  long           extra;         /* free */  
};
```

This entry is used with static analysis info. It supplies the name and kind for a few special cases not currently handled by a DNTT_SVAR, DNTT_DVAR, DNTT_TYPE, etc. It is used for a local entity that has a global scope.

Example:

If a function, has a DNTT_FUNCTION entry in the LNTT; but it can be seen globally, then a K_SA will be emitted in the GNTT, with the functions name and a base_kind of K_FUNCTION; the DNTT_XREF will follow the DNTT_SA, not the DNTT_FUNCTION.

The DNTT_SA is also used for C macros.

The XREF DNTT can be used in both the GNTT and LNTT.

7.5 Source Line Table

7.5.1 SLT Entry Format

This table consists of a series of entries, each of which is either normal, special, or assist according to the sltdesc field of the first word. Normal entries contain an address (actually a code offset relative to the beginning of the current function) and a source/listing line (by line number). Listing line numbers may be used in place of source line numbers based upon a compiler option. This will also be reflected in the DNTT_SRCFLE entries. Special entries also provide a line number (where something was declared) and point back to the DNTT which references them. This is used for quick determination of scope, including source/listing file, after an interrupt. Even if there are multiple source/listing files, all source/listing line information is accumulated in this one table.

The SLT was originally designed to be unnested, even for those languages whose LNTT must reflect their nesting. The debuggers depend upon this. For those languages that are nested the SLT must now be nested and an SLT_ASST must immediately follow each SLT_SPEC of type FUNC. The "address" field will be filled in by the compiler back-ends to point forward to the first SLT_NORM in the FUNC's scope. The "first-norm" is set to one if this SLT_NORM is the first SLT_NORM looking sequentially forward in the SLT.

The one exception to the normal/special/assist rule is the EXIT SLT. The EXIT SLT is used to identify exit points for a routine. The EXIT SLT is a special only in the sense that the sltdesc field is not equal to SLT_NORMAL. However, it contains a line number and address like a normal SLT. The EXIT SLT is used in place of a NORMAL SLT for all exit statements (such as "return" in C and FORTRAN, or the "end" of a procedure body in Pascal).

The SLT_MARKER is for use in "Chunk-Per-Som". The address field contains a new base address (replacing the current procedure's low-address field. This new base address will be added to succeeding SLT_NORMALs and SLT_EXITs to produce an absolute address.

To distinguish prologue (function setup) code emitted at the END of a function from the last line (normal SLT) of the function, a normal SLT entry with a line number of SLT_LN_PROLOGUE is used. Such SLT entries are only emitted if there is trailing prologue code, and they are always the last SLT emitted for the function except for the special SLT entry for the function END. For compilers that emit the prologue code before the main body, no special prologue SLT entry is required.

One SLT entry is emitted for (the FIRST physical line of) each executable statement, for each construct that generates a DNTT entry which points to an SLT entry, and for the prologue code, if any. The user cannot set a breakpoint without a corresponding SLT entry. Compilers must emit multiple SLT entries for parts of a composite statement (such as FOR) and for multiple statements appearing on one source line.

For compatibility, the high bits of DNTTPOINTERS in SLT entries are also set to 1, even though they are not needed here.

The global bit on DNTTPOINTERS in SLT entries should always be 0, as the LNTT contains all the scoping information.

7.5.2 SLT Types and Data Structure

Sizeof SLTTYPE is 4 bits, for a maximum of 16 possible special slttypes.

Current available SLT types are:

```
#define SLT_NORMAL 0          /* note that the field is unsigned */
#define SLT_SRCFILE 1
#define SLT_MODULE 2
```

Symbolic Debug Information

```
#define SLT_FUNCTION      3
#define SLT_ENTRY        4
#define SLT_BEGIN        5
#define SLT_END          6
#define SLT_WITH         7
#define SLT_EXIT         8
#define SLT_ASSIST       9
#define SLT_MARKER      10
#define SLT_CLASS_SCOPE  11      /* For C++ use only */

struct SLT_NORM {
    SLTTYPE      sltdesc: 4;      /* always zero */
    BITS         line: 28;      /* where in source text */
    ADDRESS      address;      /* where in function */
};                               /* two words */

struct SLT_SPEC {
    SLTTYPE      sltdesc: 4;      /* special entry type */
    BITS         line: 28;      /* where in source text */
    DNTTPOINTER  backptr;      /* where in DNTT */
};                               /* two words */

struct SLT_ASST {
    SLTTYPE      sltdesc: 4;      /* always nine */
    BITS         unused: 28;
    SLTPOINTER   address;      /* first SLT normal */
};                               /* two words */

struct SLT_GENERIC {
    unsigned long word[2];
};                               /* two words */

union sltentry {
    struct SLT_NORM      snorm;
    struct SLT_SPEC      sspec;
    struct SLT_ASST      sasst;
    struct SLT_GENERIC   sgeneric;
};                               /* two words */
```

7.6 Value Table (VT)

This table contains symbol names plus values for DNTT_CONST entries of type LOC_VT. All strings are null-terminated, as in C. There are no restrictions on the lengths of values nor the order in which they may appear. All symbol names are exactly as given by the user, e.g. there are no prepended underscores.

CONST values are not (and need not be) terminated in any way. They may be forced to word boundaries if necessary, with resulting wasted bytes.

The first byte of the table must be zero (a null string terminator), so that the null VTPOINTER results in a null name.

7.7 Ordering of Table Entries

LNTT and SLT entries must be emitted and kept in source file order wherever possible. As a minimum, named LNTT entries must be emitted and kept within the proper scope, though some compilers may emit them at the end of a scope instead of the beginning. In general, the debugger must know the emission rules for the language it is dealing with, and search the LNTT accordingly, or else always search in both directions.

Items in the GNTT are all global, so the public bit must always be set. Within the LNTT, the public bit indicates that the item is exported by the module in which it resides, and is visible within a module or procedure that imports the containing module.

Compilers and linkers are encouraged to make multiple references to DNTT, SLT, and VT entries (even chains of DNTT entries) where possible to reduce redundancy with no loss of data. They are also encouraged to emit entries grouped so that related entries are physically close, as long as no scope rules are violated.

SLT entries must be emitted in sorted line number order within each file, except for special SLT entries for ENTRYs and FUNCTIONs only. They may be out of line number order (due to nested functions, etc.) so long as the next normal SLT entry is the proper place to breakpoint the entity. For example, there can be numerous ENTRY types after a FUNCTION, all referring to the same code location. (If there are no normal SLT entries before the next FUNCTION or MODULE entry and a SLT_ASST does not immediately follow the SLT_SPEC for a FUNC, the entity has no breakpointable locations.)

SLT entries must be sorted in ascending code address order WITHIN EACH MODULE or FUNCTION body. It is impossible to require that they be sorted both by file line number and code address because function object code may be emitted or linked out of source order in a segment.

It is reasonable to expect sequential SLT entries may have the same line numbers or code locations (but not both, as that would be redundant). This might be due to multiple statements on one source line or several scope levels starting at one place in the code.

Thus, for nested languages like Pascal and Modcal, the LNTT entries must be nested to reflect the program's scope. The SLT entries should also be nested with an SLT_ASST entry following each SLT_SPEC of type FUNC.

7.8 Postprocessing

Linker postprocessing or XDB's preprocessor (**PXDB**) must be run on the debug info in the executable program file to massage the debug info so that the debugger may start up and run more efficiently.

Some of the tasks performed by PXDB are: remove duplicate global type and variable information from the GNTT, append the GNTT onto the end of the LNTT and place both back in the LNTT section, build quick look-up tables for files, procedures, modules, and paragraphs (for Cobol), placing these in the GNTT section, and reconstruct the header appearing in the header section to access this information.

- **PXDB Header and Support Data Structures:**

```
struct PXDB_header {
    int    pd_entries;    /* # of entries in function look-up table */
    int    fd_entries;    /* # of entries in file look-up table */
    int    md_entries;    /* # of entries in module look-up table */
    BITS   pxdbe : 1;     /* 1 => file has been preprocessed */
    BITS   bighdr : 1;     /* 1 => this header contains 'time' word */
    BITS   sa_header : 1; /* 1 => created by SA version of pxdb */
                                /* used for version check in xdb */

#ifdef CPLUSPLUS
    BITS   inlined : 1;    /* one or more functions have been inlined */
    BITS   spare : 12;
    short  version;        /* pxdb header version */
#else /* CPLUSPLUS */
    BITS   spare : 29;
#endif
```

Symbolic Debug Information

```
#endif /* CPLUSPLUS */
    int    globals;          /* index into the DNTT where GNTT begins */
    BITS   time;             /* modify time of file before being pxdbed */
    int     pg_entries;       /* # of entries in label look-up table */
    int     functions;        /* actual number of functions */
    int     files;           /* actual number of files */
#ifdef CPLUSPLUS
    int     cd_entries;       /* # of entries in class look-up table */
    int     aa_entries;       /* # of entries in addr alias look-up table */
    int     oi_entries;       /* # of entries in object id look-up table */
#endif
};
```

Source File Descriptor:

An element of the source file quick look-up table

```
typedef struct FDS {
    long      isym;           /* first symbol for file */
    ADRT      adrStart;       /* mem adr of start of file's code */
    ADRT      adrEnd;         /* mem adr of end of file's code */
    char      *sbFile;        /* name of source file */
    BITS      fHasDecl: 1;    /* do we have a .d file? */
    BITS      fWarned: 1;     /* have warned about age problems? */
    unsigned short ilnMac;    /* lines in file (0 if don't know) */
    int       ipd;            /* first proc for file, in PD [] */
    BITS      *rgLn;          /* line pointer array, if any */
} FDR, *pFDR;
```

Procedure Descriptor:

An element of the procedure quick look-up table

```
typedef struct PDS {
    long      isym;           /* first symbol for proc */
    ADRT      adrStart;       /* memory adr of start of proc */
    ADRT      adrEnd;         /* memory adr of end of proc */
    char      *sbAlias;       /* alias name of procedure */
    char      *sbProc;        /* real name of procedure */
    ADRT      adrBp;          /* address of entry breakpoint */
    ADRT      adrExitBp;      /* address of exit breakpoint */
#ifdef CPLUSPLUS
```

Symbolic Debug Information

```
        int          icd;          /* member of this class    */
#else /* CPLUSPLUS */
        BITS          inst;        /* instruction at entry */
#endif /* CPLUSPLUS */
#ifdef TEMPLATES
        BITS          ipd;         /* index of template for this function */
#else /* TEMPLATES */
        BITS          instExit;    /* instruction at exit */
#endif /* TEMPLATES */
#ifdef CPLUSPLUS
#ifdef TEMPLATES
        BITS          unused: 6;
        BITS          fTemplate: 1; /* function template*/
        BITS          fExpansion: 1; /* function expansion*/
        BITS          linked : 1; /* linked with other expansions*/
#else /* TEMPLATES */
        BITS          unused: 9;
#endif /* TEMPLATES */
        BITS          duplicate: 1; /* clone of another procedure */
        BITS          overloaded:1; /* overloaded function */
        BITS          member: 1; /* class member function */
        BITS          constructor:1; /* constructor function */
        BITS          destructor:1; /* destructor function */
        BITS          Static: 1; /* static function */
        BITS          Virtual: 1; /* virtual function */
        BITS          constant: 1; /* constant function */
        BITS          pure: 1; /* pure (virtual) function */
        BITS          language: 4; /* procedure's language */
        BITS          inlined: 1; /* function has been inlined */
        BITS          Operator: 1; /* operator function */
        BITS          stub: 1; /* bodyless function */
#else
        BITS          unused1: 18;
        BITS          language: 4; /* procedure's language */
        BITS          unused2: 3;
#endif
        BITS          optimize: 2; /* optimization level */
        BITS          level: 5; /* nesting level (top=0)*/
} PDR, *pPDR;
```

Module Descriptor:

An element of the module quick reference table

```
typedef struct MDS {
    long    isym;           /* first symbol for module*/
    ADRT    adrStart;       /* adr of start of mod.*/
    ADRT    adrEnd;         /* adr of end of mod.*/
    char    *sbAlias;       /* alias name of module */
    char    *sbMod;         /* real name of module*/
    BITS    imports: 1;     /* module have any imports? */
    BITS    vars_in_front: 1; /* module globals in front? */
    BITS    vars_in_gaps: 1; /* module globals in gaps? */
    BITS    unused : 29;
    BITS    unused2;        /* space for future stuff*/
} MDR, *pMDR;
```

Paragraph Descriptor:

An element of the paragraph quick look-up table

```
typedef struct PGS {
    long    isym;           /* first symbol for label */
    ADRT    adrStart;       /* memory adr of start of label */
    ADRT    adrEnd;         /* memory adr of end of label */
    char    *sbLab;        /* name of label */
    BITS    inst;           /* Used in xdb to store inst @ bp */
    BITS    sect: 1;        /* true = section, false = parag. */
    BITS    unused: 31;     /* future use */
} PGR, *pPGR;
```

Class Descriptor:

An element of the class quick look-up table for C++ support.

```
typedef struct CDS {
    char    *sbClass;       /* name of class */
    long    isym;           /* class symbol (tag) */
    BITS    type : 2;       /* 0=class, 1=union, 2=struct */
#ifdef TEMPLATES
    BITS    fTemplate : 1;  /* class template */
#endif
```

```
        BITS          expansion : 1;    /* template expansion */
        BITS          unused    :28;
#else /* TEMPLATES */
        BITS          unused    : 30;
#endif /* TEMPLATES */
        SLTPOINTER    lowscope;        /* beginning of defined scope */
        SLTPOINTER    hiscope;         /* end of defined scope */
} CDR, *pCDR;
```

Address Alias Entry:

An element of the address alias quick look-up table for C++ support.

```
typedef struct AAS {
    ADRT    low;
    ADRT    high;
    int     index;
    BITS    unused : 31;
    BITS    alternate : 1;    /* alternate unnamed aliases? */
} AAR, *pAAR;
```

Object Identification Entry

An element of the object identification quick look-up table for C++ support.

```
typedef struct OIS {
    ADRT    obj_ident;        /* class identifie */
    long    isym;             /* class symbol */
    long    offset;           /* offset to object start */
} OIR, *pOIR;
```

7.9 Debug Format Changes for Debugging of Optimized Code (DOC)

7.9.1 Debug Format Changes

The following describes the changes to the debug format for HP-UX 10.0. The primary change to the debug format is the addition of a new debug space and debug subspaces. For code compiled with -g and -O, the debug information will be generated into a new space named \$PINFO\$ (after processing with pxdB -- prior to pxdB processing the

debug information will be generated into the \$DEBUG\$ space). All the standard xdb-format subspaces will be placed into the \$PINFO\$ space along with the two new subspaces: \$LINE\$, and \$LT_OFFSET\$.

7.9.2 Object File Format Details

When a file is compiled with debug and optimization specified, the compilers build a \$DEBUG\$ space and a \$LT_OFFSET\$ subspace within the \$DEBUG\$ space. The linker builds the \$LINE\$ subspace within the \$DEBUG\$ space when debug info is seen in any object file being linked. When pxdB processes the executable file, the \$DEBUG\$ space is renamed \$PINFO\$ if it detects DOC format debug info in the file.

The compilers supply line table information to the linker in the form of fixup requests. Two new fixup requests have been defined to signal the building of DOC line tables (which includes information to be included in the first entry of the line table) and to generate special line table escape entries. The information for the line number tables is supplied by the fixup request, R_LINETAB.

The compilers generate the R_LINETAB fixup to request that DOC line tables be built. This fixup passes in a version number and subspace index and subspace offset of a location which must be patched with the offset of the line table which is about to be built. The R_LINETAB fixup request is a 9-byte with the following fields:

Offset	Length	Field
0	1	R_LINETAB
1	1	version number
2	3	symbol index (symbol-relative loc to patch w/line table offset)
5	4	offset (symbol + offset = location to patch w/line table offset)

The first parameter is a 1-byte version number which identifies the line table version (format). The actual value is not important to the linker. The second parameter is a symbol index to be used in conjunction with the third parameter, an offset, as a location which is to be filled with the offset (relative to the \$LINE\$ subspace) of the line table about to be built.

The line number information is passed to the linker via the R_STATEMENT fixup request, which is embedded within the fixups for the code at statement boundaries. The R_STATEMENT fixup has three variants to handle one, two and three byte statement of line numbers as necessary. The actual meaning assigned to the number, whether it be statement numbers or line numbers, is irrelevant to the linker, and needs to be agreed upon only by the compiler and the end user of the line table information.

The R_LINETAB_ESC fixup is a 3-byte fixup defined as follows:

Offset	Length	Field
0	1	R_LINETAB_ESC (0xDB)
1	1	escape code
2	1	number of following R_STATEMENT fixups containing escape data.

This fixup request is used to place escape entries into the line table. There are several escape entries defined in the line table format which are used by the debugger and other tools when processing the line table. Some of these escapes must be generated by the linker, the others are generated by the compiler and the linker does not need to know the details of these escapes. The escapes entries which are not generated by the linker are entered into the line table via a combination of the R_LINETAB_ESC and R_STATEMENT fixups.

The first parameter contains the actual escape code which is to be placed into a 1-byte entry in the line table. The second parameter specifies how many of the following R_STATEMENT entries contain data to be entered directly into the line table (these statement fixups will not contain line numbers -- instead they hold data which is to be placed directly into the line number table as part of the escape sequence). With the currently defined escapes, the value of the second parameter will be in the range [0,4].

7.9.3 Building the Line Tables

All line tables will be placed into the \$LINES\$ subspace of the executable file. The linker must create a new line table each time an R_LINETAB fixup is processed. If a line table is in progress then it must be completed by entering a dst_in_end escape and the final pc delta entry. Each line table is terminated when 1) a new R_LINETAB fixup is seen or 2) when the end of the current code subspace is reached.

When a new line table is started the version number passed as a parameter to the fixup is used as the first one-byte entry in the table. The symbol and offset parameters passed in the R_LINETAB fixup must be saved along with the corresponding line table offset as 'fixups' to be applied to the symbol+offset location when that symbol's subspace is processed. The first R_STATEMENT entry processed after an R_LINETAB fixup will generate a four-byte entry containing the absolute code address associated with the fixup.

The size of the next entry (in bytes) is determined by the absolute line number value (the value passed to the R_STATEMENT fixup). The linker must emit the absolute line number into the table using the minimum number of bytes required by the line number value. For example, if the line number is less than 256, then the absolute line number entry will be one byte; if the line number is greater than 255 and less than 65536, then the line number entry will be 2 bytes, etc. Each subsequent R_STATEMENT entry will cause one entry (consisting of one or more bytes) to be generated into the line table.

The linker needs to be aware of, and generate, some of the escape codes defined for the DOC line table. These escape codes are as follows:

Table entry for ESC	ESC Name	Description
	dst_ln_end	end escape; final entry follows. The final entry contains the code size of the last statement in high 4 bits (i.e. the last PC delta); the low 4 bits are 0
	dst_ln_pad	This byte is padding
	dst_ln_dpc1_dln1	The next table entry is a one byte pc delta followed by a one byte line delta.
	dst_ln_dpc2_dln2	The next table entry is a two byte pc delta followed by a two byte line delta.
	dst_ln_pc4_ln4	The next table entry is a four byte absolute pc followed by a 4-byte absolute line number.
	dst_ln_dpc0_dln1	The next table entry is a one byte line delta; the pc delta is zero.

The linkers must use the multi-byte line delta and pc-delta escapes whenever the line or code delta values exceed the range that can be expressed in a 1-byte entry. There must be one line table entry to express each R_STATEMENT fixup. For example, if either the line number delta or code delta falls outside of the range [-8,7] (line delta) or [0,11] (code delta), but is less than the byte-range [-127,128] (line delta) or [0,255] (code delta) then a **dst_ln_dpc1_dln1** must be generated. Similarly, if the delta range for either line or code delta cannot be described in the 1-byte format, then a **dst_ln_dpc2_dln2** (two-byte line and pc delta escape format) must be used. Finally, if the code and line deltas exceed the 2-byte format, then a **dst_ln_pc4_ln4** absolute line number and code address must be used.

The R_LINETAB_ESC fixup directs the linker to treat <n> following R_STATEMENT fixups as absolute data entries. The R_LINETAB_ESC causes the linker to generate a one-byte entry into the table which contains the data value passed in as the first argument. The second R_LINETAB_ESC argument specifies the number of following R_STATEMENT entries which contain data to be directly entered into the line-table. These R_STATEMENT entries will not cause the normal pc-delta/line-delta entries to be generated; Instead, the argument passed to the R_STATEMENT fixups will be used as absolute data for a one-byte entry in the line table.

7.9.4 Debug Format Changes

The new subspace, \$LT_OFFSET\$, will be placed into the \$DEBUG\$ space by the compilers when optimization is specified with debug (-g and -O). The format and the \$LT_OFFSET\$ table is a list of 1-word entries; each entry contains a line table offset

which corresponds to the beginning of each line table in the \$LINES\$ subspace (in order). One line table will be emitted for each NTT_FUNC debug entry. This subspace is temporary for UX10.0 xdb-DOC transition and will be obsoleted in post-UX10.0 releases.

Xdb-style \$GNTT\$, \$LNTT\$, \$SLT\$ and \$VT\$ will be placed into the \$DEBUG\$ space by the compilers when optimization and debug are specified together on the command line. If no optimization is requested (plain -g) then the standard xdb debug information will be generated into the \$DEBUG\$ debug space.

The xdb-style \$HEADER\$ subspace will be modified to include new fields when both debug and optimization are specified (the \$DEBUG\$ space and xdb format will be unchanged when -g is used without optimization, or when static analysis (-y) is used).

The DOC information header is defined as follows:

```
struct DOC_info_header {
    unsigned int xdb_header: 1;    /* bit set if this is post-3.1 xdb */
    unsigned int doc_header: 1;    /* bit set if this is doc-style header */
    unsigned int version: 8;       /* version of debug/haer format. For 10.0
                                   the value will be 1 */
    unsigned int reserved_for_flags: 20; /* for future use; -- must be set to 0 */
    unsigned int has_lines_table: 1; /* space contains a $LINES$ subspace for
                                   ine tables.*/
    unsigned int has_lt_offset_map: 1; /* space contains an lt_offset subspace
                                   for line table mapping */

    long gntt_length;
    long lntt_length;
    long slt_length;
    long vt_lenth;
    long xt_length;
}
```

Similarly, the pxdB header must be modified to include the DOC fields when emitted into the \$PINFO\$ space of an executable.

The DOC pxdB header is defined as follows:

```
struct DOC_info_PXDB_header {
    unsigned int xdb_header: 1;    /* bit set if this is post-3.1 xdb */
    unsigned int doc_header: 1;    /* bit set if this is doc-style header */
    unsigned int version: 8;       /* version of debug/haer format. For 10.0
                                   the value will be 1 */
    unsigned int reserved_for_flags: 20; /* for future use; -- must be set to 0 */
    unsigned int has_lines_table: 1; /* space contains a $LINES$ subspace for
                                   ine tables.*/
}
```

Symbolic Debug Information

```
unsigned int has_lt_offset_map: 1;      /* space contains an lt_offset subspace
                                         for line table mapping */

int    pd_entries;      /* # of entries in function look-up table */
int    fd_entries;      /* # of entries in file look-up table */
int    md_entries;      /* # of entries in module look-up table */
BITS   pxdbe: 1;        /* 1 => file has been preprocessed */
BITS   bighdr: 1;        /* 1 => this header contains 'time' word */
BITS   sa_header: 1;     /* 1 => created by SA version of pxdbe */
                           /* used for version check in xdb */

BITS   inlined: 1;       /* one or more functions have been inlined */
BITS   spare: 12;

short  version;          /* pxdbe header version */
int    globals;          /* index into the DNTT where GNTT begins */
BITS   time;             /* modify time of file before being pxdbe */
int    pg_entries;       /* # of entries in label look-up table */
int    functions;        /* actual number of functions */
int    files;            /* actual number of files */
int    cd_entries;       /* # of entries in class look-up table */
int    aa_entries;       /* # of entries in addr alias look-up table */
int    oi_entries;       /* # of entries in object id look-up table */

};
```

For example, the \$PINFO\$ debug space will contain the following subspaces when -g and -O are specified together on the command line:

```
$PINFO$
$HEADER$
$GNTT$
$LNTT$
$SLT$
$VT$
$LINE$
$LT_OFFSET$
```

Note: there will be no \$XT\$ table for static analysis. Static analysis (-y) and optimization is incompatible for UX10.0.

7.9.5 Line Number Table Definition

The line number definition is based on the DST (Domain DDE Symbol Table) .lines definition. Although there were changes to support additional escape codes, all existing DST .lines escapes have been retained. The line table format is defined as a stream of nibble pairs, where the first nibble represents a PC delta, and the second a line number delta. The PC delta is unsigned, and runs from 0..15. The PC delta values 12..15 are used for special escape handling. PC delta values 12 and 13 are used to signify short-form context switches. PC delta values 14 and 15 are used to signify two sets of 16 escape codes in the line number delta field.

PC Delta	Line Delta or Bit #	Interpretation
0..11	-8..7	interpreted as line delta
12	4 bits - rrcc	interpreted as context switch: rr (0..3) is run length in entries, interpreted as 1..4. cc (0..3) is context index number.
13	4 bits - rrrd	interpreted as context switch: rr (0..3) is run length in entries, interpreted as 1..4. dd (0..3) is context index number minus 4, so is interpreted as contexts 4..7.
14	0..15	interpreted as new escape codes (set #2)
15	0..15	interpreted as DST escape codes (set #1).

Note that it is possible to have entries with PC deltas of 0. This will be used to associate multiple source lines to a single block of code.

The image table is a structure intended to be used in the presence of inlining. It defines the full source file context of inlined code. It also provides a starting line number from which subsequent line number deltas are applied, within the life of the run length. The code which interprets the delta stream will obtain the starting line number for a context from the image table at the first encounter of it's index number. The interpreter must then maintain a running count of the context's current line number. Note that a context switch does not signify creation of a line number table entry, but rather is used to set up the context to which subsequent deltas are applied.

The PC and line number bases to which subsequent deltas are applied are set forth in the table via special escapes. This escape and starting bases must appear prior to any applicable delta pairs, and the bases may be reset at any time. Once the bases are set, the interpreter will generate a line number table entry whenever it encounters a PC/number delta pair (which may take 1, 2, 5 or 9-byte forms).

Symbolic Debug Information

Set #1 Escape codes (same as DST)

decimal	name	function
0	dst_ln_pad	pad type
1	dst_ln_file	pad byte fill escape
2	dst_ln_dpc1_dln1	1 byte pc delta, 1 byte line delta
3	dst_ln_dpc2_dln2	2 byte pc delta, 2 byte line delta
4	dst_ln_pc4_ln4	4 byte absolute pc number, 4 byte abs. line
5	dst_ln_dpc0_dln1	pc delta = 0, 1 byte line delta
6	dst_ln_ln_ff_1	statement escape, stmt # = 1
7	dst_ln_ln_off	statement escape, stmt # = next byte
8	dst_ln_entry	entry escape, next byte is entry number
9	dst_ln_exi	exit escape
10	dst_ln_stmt_end	gap escape, 4 bytes pc delta
11	dst_ln_escape_11	reserved
12	dst_ln_escape_12	reserved
13	dst_ln_escape_13	reserved
14	dst_ln_nxt_byte	next byte contains real escape code
15	dst_ln_end	end escape, final entry follows

Set #2 Escape Codes (additional to DST ones)

decimal	name	function
0	dst_ln_ctx_1	next byte describes context switch with 5-bit index into the image table and 3-bit run length. If run length is 0, the context is considered active until context end switch or new context switch are encountered.
1	dst_ln_ctx_2	next 2 bytes describe context switch with 13-bit index and 3-bit run length. If run length is 0, the context is considered active until context and switch or new context switch are encountered.
2	dst_ln_ctx_4	next 4 bytes describe context switch with 29-bit index and 3-bit run length. If run length is 0, the context is considered active until context and switch or new context switch are encountered.
3	dst_ln_ctx_end	end current context.
4	dst_ln_col_run_1	next byte is a column position marking the beginning of the next statement, following byte is length of statement.
5	dst_ln_col_run_2	next 2 bytes are a column position marking the beginning of the next statement, following two bytes are length of a statement.
6	dst_ln_init_base1	next 4 bytes are an absolute PC base address. Immediately following is a 1-byte starting line number.
7	dst_ln_init_base2	next 4 bytes are an absolute PC base address. Immediately following is a 2-byte starting line number.
8	dst_ln_init_base3	next 4 bytes are an absolute PC base address. Immediately following is a 3-byte starting line number.
9-15	reserved	for future use.

7.9.6 View/modify globals and arguments when safe

Globals may be set and viewed safely at procedure entry and exit for C, C++ and FORTRAN code. For Pascal, however, the Pascal front end is intelligent enough to recognize some instances in which a global may be safely promoted across procedure boundaries. Thus, viewing and setting of globals must be considered unsafe at all times for Pascal code. Arguments may be viewed and set safely at procedure entry for C, C++, FORTRAN and Pascal. The 10.0 functionality will permit setting and viewing of globals and arguments at unsafe times, but DDE will generate a warning of unreliability for these

operations. Further, setting of locals will also be allowed, but will always cause a warning.

Symbolic Debug Information

Stack Unwind Library

8.1 Overview

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, *C++ exception handling* also require stack unwinding capabilities.

The stack unwind information is generated once at compile time via fixups 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.

8.2 Requirements for 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 following the procedure calling conventions.

To guarantee that a routine is unwindable, the assembly programmer should strictly adhere to the stack and register usage conventions described in the Run-time Architecture document. 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.

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 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:

word #1	.PROC (start address of the procedure)
word #2	.PROCEND (end address of the procedure)
word #3	.CALLINFO (unwind descriptor)
word #4	

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.

8.2.1 Unwinding Across an Interrupt Marker

Information such as machine state (i.e., register contents) are pushed on stack when interrupt or trap occurs. This area of stack is called the interrupt marker and is different from the normal stack marker. The routine *_sigreturn()* marks the interrupt marker by having the *HP_UX_interrupt_marker* bit of its unwind descriptor set. Unwind tool must check this bit when unwinding through each frame. When the *HP_UX_interrupt_marker* bit is set, register contents must be restored from the interrupt marker. The interrupt marker is defined in the *sigcontext* structure.

8.2.2 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 as described in Section 6.3.16 on page 167.

In the HP-UX shared library implementation, GR 19 points to a shared library descriptor. This descriptor contains a pointer to the location where the unwind tables and the stub tables are located. Each shared library has its own tables.

When unwinding through the HP-UX export stub, the PC return register (RP) and GR 19 are restored from the stack (SP-24 for RP and SP-32 for GR 19).

8.2.3 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.

8.2.4 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 (GR 27) is modified. Note that DP should never be altered by user code, only by system code as is absolutely necessary.

8.2.5 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 do not 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 word immediately following the last callee-saves general register.

8.2.6 Sample entry and exit 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 .. gr5` and `fr12 .. fr15` get stored in the stack. It also allocates 24 bytes of space for local variables. The entire frame size including the frame marker is 128 bytes. Note that this is the exact sequence of code that should be happening for procedure entry and exit, the unwinding utilities may fail if an interrupt occurs on an instruction in a non-standard entry or exit sequence.

```
.SPACE $TEXT$
```

```
.SUBSPA $CODE$,QUAD=0,ALIGN=4,ACCESS=44,CODE_ONLY,SORT=24
```

initboard

.PROC

.CALLINFO CALLER,FRAME=24,ENTRY_FR=15,ENTRY_GR=5,SAVE_RP

.ENTRY

STW rp,-20(sp)

FSTD,MA fr12,8(sp)

FSTD,MA fr13,8(sp)

FSTD,MA fr14,8(sp)

FSTD,MA fr15,8(sp)

STWM r3,96(sp)

STW r4,-92(sp)

STW r5,-88(sp)

:: procedure body

LDW -88(sp),r5

LDW -92(sp),r4

LDWM -96(sp),r3

FLDDS,MB -8(sp),fr15

FLDDS,MB -8(sp),fr14

FLDDS,MB -8(sp),fr13

BV r0(rp)

.EXIT

FLDDS,MB -8(sp),fr12

.PROCEND ;

8.3 Role of Stubs in Unwinding

The stub unwind region (also called the linker stub table) contains unwind descriptors for linker-generated stubs. Stubs are usually 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 is 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. If a procedure needs to return through a parameter relocation stub, the unwind mechanism needs to know that the extra `rp` value is saved in the stack marker. If execution is stopped in the middle of a stub, unwind needs to know that, especially if inside a parameter relocation stub, where the stack pointer may have been bumped by 8 bytes to create a temporary storage area. The stub-unwind descriptors have the following format:

```
struct stub_desc {  
  
    unsigned int    addr;                /* address of the first instruction of the stub */  
  
    unsigned int    mbz1: 4;             /* must be zero - reserved */  
  
    unsigned int    type: 4;             /* stub type */  
  
    unsigned int    mbz2: 3;             /* must be zero - reserved */  
  
    unsigned int    relocen: 5;          /* used only for parameter relocation stubs;  
                                         contains the number of the instruction which  
                                         stores RP on the stack in the stub. */  
  
    unsigned int    length: 16;          /* length (# of words) of stub area */  
  
};
```

In some cases, a contiguous sequence of calling, called, or long branch stubs or milli-code 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:

\$UNWIND_START\$	at beginning of unwind table
\$UNWIND_END\$	at beginning of stub table

\$RECOVER_START\$	at beginning of recover table
\$RECOVER_END\$	at end of recover table

The three tables mentioned above, namely the unwind, stub and recover tables, are required to be contiguous. In a shared library, the DLT slot at 0 (gr 19) contains a self-relative pointer to a four-word descriptor containing the four pointers to the unwind tables corresponding to the four symbols above.

8.3.1 The Stub Unwind Types

The following table describes the stub-unwind types in bits 4..7 of the second word of the two-word unwind descriptors for stubs:

Stub Names	Value	Description
NULL	0	not used
LONG_BRANCH_STUB	1	stubs generated for branches beyond 256K-bytes offset.
LOCAL_RELOC_STUB	2	parameter relocation stub
EXTERN_IMPORT_STUB	3	MPE shared library import stub
EXTERN_EXPORT_STUB	4	calls entry point, handles inter-quad return; deposits caller's exec level in rp
LONG_LOAD_STUB	5	not used
HPUX_IMPORT_STUB_NO_RP	6	signal to the unwind library and all other unwind users that this is an HP-UX shared library import stub (in either a program file or a shared library) that does not save RP before branching to the callee.
MILLILONG_BRANCH_STUB	7	like LONG_BRANCH_STUB, used to reach millicode routines
INTERQUAD_IMPORT_STUB	8	loads r22 with address of routine in quad0 and branches to _sr4export
HPUX_EXPORT_STUB_NO_RP	9	signal to the unwind library and all other unwind users that this is an HP-UX shared library export stub (in either a program file or a shared library) that does not save RP before branching to the entry point; and in fact does not trap the return from the entry before control passes back to the caller.
HPUX_EXPORT_STUB	10	HP-UX shared library export stub, like DL_EXPORT_STUB

Stub Names	Value	Description
HPUX_IMPORT_STUB	11	HP-UX shared library import stub used in an incomplete executable. Loads r19 required by callee, makes inter-quad branch, stores rp at -24 (sp).
SHLIB_IMPORT_STUB	12	HP-UX shared library import stub used in a shared library, see HPUX_IMPORT_STUB.
LONG_SHLIB_IMPORT_STUB	13	like SHLIB_IMPORT_STUB except PLT entry is too far from r19.
SHL_LONG_BRANCH_STUB	14	PC-relative long branch stub used in a shared library.
FDP_COUNTING_STUB	15	Stub generated to count branches for feedback directed positioning.

8.3.2 Unwinding from Parameter Relocation Stub

A parameter relocation stub creates its own temporary 8-byte stack frame while it's executing, so the stack unwind mechanism needs to understand where the stack pointer gets incremented and decremented. There are two forms of parameter relocation stub. The first form saves rp and catches the return path so it can relocate the return value; the second form is one way, so it does not have to save rp.

In the following two examples, assembly code are shown to handle arg1/arg0 to farg1 and arg3/arg2 to farg3 parameter relocation.

With return relocation:

```

a:                                     ; relocate parameters
    stws,ma    arg1, 8(sp)           ; create temporary stack frame
    stws       arg0, -4(sp)
    fdds       -8(sp), farg1
    stws       arg3, -8(sp)
    stws       arg2, -4(sp)
    fdds,mb    -8(sp),farg3         ; destroy temporary frame

b:                                     ; save rp and call the function
    stw        rp, -8(sp)
    bl,n       func,rp
    nop

                                     ; function returns here
                                     ; relocate the return value

c:
    fstds, ma   fret0, 8(sp)         ; create temporary stack frame
    ldws       -4(sp), ret1
    ldws, mb    -8(sp), ret0         ;destroy temporary frame

d:
    ldw        -8(sp), rp
    bv,n       0(rp)

e:

```

Without return relocation:

```

a:                                     ; relocate parameters
    stws,ma    arg1, 8(sp)           ; create temporary stack frame
    stws       arg0, -4(sp)
    fdds       -8(sp), farg1
    stws       arg3, -8(sp)
    stws       arg2, -4(sp)
    fdds,mb    -8(sp),farg3         ; destroy temporary frame

b:                                     ; branch to the function
    b,n       .+8

c:
d:
e:

```

For parameter relocation stubs (type 2) and export stubs (types 9 and 10), the size (in instructions) of the argument relocation code $((b - a)/4)$ is recorded in the *reloclen* field, and the total size (in instructions) of the stub $((e - a)/4)$ in the *length* field.

If there is no return relocation path, the value $(length - reloclen)$ will be 1.

If $(length - reloclen)$ is greater than 1, the relative positions of the labels *c* and *d* can be inferred from the values of *reloclen* and *length* as follows:

$$c = b + 12 = a + (reloclen * 4) + 12$$

$$d = e - 8 = a + (length * 4) - 8$$

When unwinding, use the following table to determine how to find the next frame, based on the current *pc*:

pc	return ptr	psp
pc == a	rp	sp
a < pc < b	rp	sp - 8
pc == b	rp	sp
b < pc <= c	-8(sp)	sp
c < pc < d	-8(psp)	sp - 8
d <= pc < e	-8(sp)	sp

For export stubs (types 9 and 10), the calculations of positions *c* and *d* are different because the stubs are different:

$$c = b + 8 = a + (reloclen * 4) + 8$$

$$d = e - 16 = a + (length * 4) - 16$$

8.4 External Interface

8.4.1 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.

Following 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 reserved: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 sr4export:1;
    unsigned int cxx_info:1;
    unsigned int cxx_try_catch:1;
    unsigned int sched_entry_seq:1;
        unsigned int reserved1:1;
    unsigned int Save_SP:1;
    unsigned int Save_RP:1;
    unsigned int Save_MRP_in_frame:1;
    unsigned int save_r19:1;
    unsigned int Cleanup_defined:1;
    unsigned int MPE_XL_interrupt_marker:1; /* Word 4 */
    unsigned int HP_UX_interrupt_marker:1;
    unsigned int Large_frame_r3:1;
    unsigned int alloca_frame:1;
        unsigned int reserved2:1;
    unsigned int Total_frame_size:27;
};
```

region_start

This is the starting address of the unwind region.

region_end

This is the end address of the unwind region.

word # 3 and word #4: Flags

The 3rd and the 4th word of the unwind descriptor contains bit flags and stack frame size that are used by the unwind utility routines. The number in the following brackets are only used for identifying purpose.

1. **Cannot_unwind** (bit 0): One (1) 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** (bit 1): One if this region is a millicode routine; zero otherwise.
3. **Millicode_save_sr0** (bit 2): One if this (millicode) routine saves sr0 in its frame (at current_SP - 16); zero otherwise.
4. **Region_description** (bits 3-4): 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** (bit 6): One if the sole entry-save space register sr3 is saved/restored by the associated entry/exit code sequence; zero otherwise.
6. **Entry_FR** (bit 7-10): The number of entry-save floating-point registers saved/restored by the associated entry/exit code sequence.
7. **Entry_GR** (bit 11-15): 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** (bit 16): 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** (bit 17): 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** (bit 18): 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** (bit 19): Indicates the associated region is a special millicode routine which implements the Ada frame extension operation.
12. **Stack_Overflow_Check** (bit 20): Indicates the associated region has an Ada stack overflow check in its entry sequence(s).
13. **Two_Instruction_SP_Increment** (bit 21): 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. **sr4export** (bit 22): Indicates hand written sr4 export stub.
15. **cxx_info** (bit 23): This bit is used to indicate one or both of the followings:
 - (a) the associated function or region has a C++ exception specification.
 - (b) the associated function or region has objects which might require cleanup (destruction).
16. **cxx_try_catch** (bit 24): This bit is used to indicate that the associated function or region has one or more C++ try/catch constructs.
17. **sched_entry_seq** (bit 25): This bit indicates optimizer may have scheduled entry code. U_get_previous frame emits a warning message in this case indicating that context restoring unwind is not possible.
18. **Save_SP** (bit 27): 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.
19. **Save_RP** (bit 28): 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.
20. **Save_MRP_in_frame** (bit 29): 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.
21. **Save_r19** (bit 30): One if gr19 is saved for shared library tables.
22. **Cleanup_defined** (bit 31): The interpretation of this field is dependent upon the language processor which compiled the routine.
23. **MPE_XL_interrupt_marker** (bit 32): One if the frame layout corresponds to that of an MPE XL interrupt marker.
24. **HP_UX_interrupt_marker** (bit 33): One if the frame layout corresponds to that of an HP-UX interrupt marker.

- 25. *Large_frame_r3*** (bit 34): One if gr3 is changed during the entry sequence to contain the address of the base of the (new) frame.
- 26. *alloca_frame*** (bit 35): This bit is set if `alloca()` is used and has been inlined. This indicates gr3 or gr4 may contain the previous sp value.
- 27. *Total_frame_size*** (bit 37-63): 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.

8.4.2 Unwind Utility Routines

The unwind utility routines currently reside in the `libcl.a` (`libcl.sl` for shared library). The following section describes these routines and their interfaces.

- **`U_get_unwind_table`**

```
struct utable {  
    unsigned unwind_table_start;  
    unsigned unwind_table_end;  
};
```

```
struct utable U_get_unwind_table(unsigned int dp_value);
```

This routine 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.

ARG0: DP value of routine being unwound to. (only used on MPE/iX)

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.

- **`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, this routine 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.

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.

- **U_get_previous_frame**

```
struct current_frame_def {  
  
    unsigned curr_frame_size;      /* Frame size of current routine. */  
    unsigned curr_sp;              /* The current value of stack pointer. */  
    unsigned curr_pcspace;         /* PC-space of the calling routine. */  
    unsigned curr_pcoffset;        /* PC-offset of the calling routine. */  
    unsigned curr_dp;              /* Data Pointer of the current routine. */  
    unsigned curr_rp;              /* Initial value of RP. */  
    unsigned curr_mrp;             /* Initial value of MRP. */  
    unsigned curr_sr0;             /* Initial value of sr0. */  
    unsigned curr_sr4;             /* Initial value of sr4 */  
    unsigned r3;                   /* Initial value of gr3 */  
    unsigned cur_r19;              /* GR19 value of the calling routine,  
                                   used only in shared library HP-UX. */  
  
    int r4;                        /* for alloca run-time stack memory  
                                   allocation */  
  
    reserved;                      /* may have values in future releases */  
  
};  
  
struct previous_frame_def {  
  
    unsigned prev_frame_size;      /* frame size of calling routine. */  
    unsigned prev_sp;              /* SP of calling routine. */  
    unsigned prev_pcspace;         /* PC_space of calling routine's caller. */  
    unsigned prev_pcoffset;        /* PC_offset of calling routine's caller. */  
    unsigned prev_dp;              /* 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 uend;                 /* end of the unwind region. */  
    unsigned uw_index;             /* index into the unwind table. */  
};
```

```
    unsigned prev_r19;          /* GR19 value of the caller's caller. */
    int r3;                    /* value for gr3, for run-time-stack
                               memory allocation */
    int r4;                    /* value for gr4, for run-time-stack
                               memory allocatoin */

};
```

```
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, this routine returns 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 return value of this function means:

0: normal;

Negative:

-1: if curr_pcspace, curr_pcoffset 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: The frame is not unwindable for some reason.

1: no unwind_descriptor

0x7fffffff: cannot_unwind bit on in previous unwind descriptor

Assembly interface:

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 frame size, current sp, current pc offset, current dp, r3, and current r19 values from the *prev_frame* into the corresponding fields in the *curr_frame* 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, *curr_frame_size* should be zero on the initial call.

- **U_get_previous_frame_x**

```
int U_get_previous_frame_x (
    struct current_frame_def *curr_frame;
    struct previous_frame_def *prev_frame;
    int size);
```

The functionality of this routine is the same as *U_get_previous_frame*. The only difference is the addition of the third parameter. This routine is introduced to allow for new fields to be added to the *current_frame_def* and *previous_frame_def*. With the *alloca* support, the data structures have to be extended to include new fields for *alloca* run-time stack memory information. The *size* field is used to specify the number of bytes used for the *previous_frame_def*. Starting at 10.0, users should start using *U_get_previous_frame_x* instead of *U_get_previous_frame* to access to the previous frame on stack.

- **U_get_recover_table**

```
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). */
};

struct rtable {
    unsigned recover_table_start;
    unsigned recover_table_end;
};
```

```
struct rtable U_get_recover_table (unsigned int dp_value);
```

This function returns the code offsets of the start and end of the recover table of a given object module.

This routine and the one describes below (*U_get_recover_address*) 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* statement.

The recover table has three word entries containing the beginning and the end addresses of the unwind region and the resume address. It is word-aligned.

This function 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.

- **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, an -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.

- **U_STACK_TRACE**

`U_STACK_TRACE();`

Applications can obtain stack traces easily using the `U_STACK_TRACE()` routine. 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.

- **`U_get_shLib_text_addr`**

`int U_get_shLib_text_addr(int GR19);`

Given the GR 19 value, this routine will return -1 if the corresponding code is not in the HP-UX shared library, otherwise it will return the text address of the shared library.

- **`U_get_shLib_unw_tbl`**

`struct utable U_get_shLib_unw_tbl(int GR19);`

Given the GR 19 value, this routine will return -1 if the corresponding code is not in the HP-UX shared library, otherwise it will return the address of unwind start and unwind end of the shared library.

- **`U_get_shLib_recv_tbl`**

`struct rtable U_get_shLib_recv_tbl(int GR19);`

Given the GR 19 value, this routine will return -1 if the corresponding code is not in the HP-UX shared library, otherwise it will return the address of recover start and recover end of the shared library.

- **`U_init_frame_record`**

`void U_init_frame_record(struct current_frame_def* cfi);`

This routine initializes the fields in the current frame def as follows:

<code>cur_frsize</code>	this frame's size
<code>cursp</code>	this frame's sp
<code>currfs</code>	this frame's space id (program counter's space)
<code>currlo</code>	this frame's pc offset (an instruction address within the routine <code>U_init_frame_record</code> .) This field is later set to the return link

offset to the calling (or previous) frame

by a call to `U_prep_frame_rec_for_unwind`.

`curdp` this frame's data pointer

`toprp` the value in `rp`

`topmrp` this field is zeroed

`topsr0` not initialized

`topsr4` not initialized

`r3` the value in general register `r3`

`r19` the value in general register `r19` (e.g. this frame's

shared library table pointer). This field is later set

to the calling frame's shared library table pointer by a

call to `U_prep_frame_rec_for_unwind`.

`r4` the value in general register `r4`.

- **`U_prep_frame_rec_for_unwind`**

```
void U_prep_frame_rec_for_unwind(struct current_frame_def* cfi);
```

This routine initializes the `curRLO` and `cur_r19` fields in the `current_frame_def` record to the following:

`curRLO`: the return link offset into the calling routine.

`cur_r19`: the calling routines `r19` (shared library table pointer) value
if it is shared code.

- **copy_frame_info**

void

copy_frame_info(struct current_frame_def *cfi, struct previous_frame_def *pfi)

This routine copies pertinent fields from the previous frame record to the current frame record in preparation for the next call to U_get_previous_frame_x.

The parameters are:

cur_frsize, cursp, currls, currlo, curdp, r3, r4, and cur_r19.

8.4.3 Initialize a Stack Unwind

The best way to initialize a Stack Unwind is the following

```
#include "unwind_headers.h"
```

```
main()
```

```
{
```

```
    struct current_frame_def cfd;
```

```
    struct previous_frame_def pfd;
```

```
    int condition;
```

```
    U_init_frame_record(&cfd)
```

```
    U_prep_frame_rec_for_unwind(&cfd)
```

```
    while (condition)
```

```
    {
```

```
        U_get_previous_frame_x(&cfd,&pfd,52);
```

```
        /* code for calculations, symbol lookups, etc. on info in &pfd */
```

```
        .
```

```
        .
```

```
        .
```

```
        copy_frame_info(&cfd,&pfd);
```

```
    }  
}
```

8.4.4 Unwind Examples: Using `U_get_previous_frame`

This following example illustrates how to make use of the `U_get_previous_frame` routine to write a stack trace into a character string. This example demonstrates that trace mechanism works for both archived and shared library routines.

Since a full stack trace requires access to the symbol tables in the program file, we have omitted the symbols from the output.

To try out the example, do:

```
make unwind_example
```

`U_get_previous_frame` 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 `curr_frame` to the appropriate machine state, and then copying the first five `prev_frame` fields into the corresponding fields for successive calls, until end of stack is reached. The initial set-up of `curr_frame` is done using a supported low-level routine which HP has written. This assembly level routine **MUST** be in the same image as the routine which uses it. This is the method we recommend using when priming the initial `curr_frame` for `U_get_previous_frame`. NOTE: the `curr_frame` and `prev_frame` fields are subject to change across releases. Thus, you should always extract the low-level routine from the system on which the executable will be built for (the location of the routine is explained in `trace.c` below).

```
=====  
Makefile  
=====
```

```
unwind_example: example.out example.sl trace.sl output.txt
```

```
output.txt: example.out  
             example.out 1>output.txt 2>&1
```

```
example.out: test_unwind.c example.sl trace.sl  
             cc -Aa -o example.out test_unwind.c example.sl trace.sl -lcl  
example.sl: test_shl.c  
            cc -c -Ae +z test_shl.c  
            ld -o example.sl -b test_shl.o  
            rm test_shl.o
```

```
trace.sl: trace.c unwind.h ugetfram.s  
          cc -c -Aa +z trace.c ugetfram.s  
          ld -o trace.sl -b trace.o ugetfram.o  
          rm trace.o ugetfram.o
```

```
clean:  
      rm -f *.o *.out *.sl *.txt
```

```

=====
unwind.h
=====

#ifndef UNWIND_HEADER_FILE
#define UNWIND_HEADER_FILE

typedef struct cframe_info {
    unsigned cur_fsize;      /* frame size */
    unsigned cursp;          /* stack pointer */
    unsigned currls;         /* PC-space of CALLING routine */
    unsigned currlo;         /* PC-offset of CALLING routine */
    unsigned curdp;          /* data pointer */
    unsigned toprp;          /* return pointer */
    unsigned topmrp;         /* millicode return pointer */
    unsigned topsr0;         /* sr0 */
    unsigned topsr4;         /* sr4 */
    unsigned r3;             /* gr3 */
    unsigned cur_r19;        /* linkage-table pointer (gr19) - for PIC code */
} cframe_info;

typedef struct pframe_info {
    unsigned prev_fsize;     /* frame size */
    unsigned prevsp;         /* stack pointer */
    unsigned prevrls;        /* PC-space of CALLING routine */
    unsigned prevrlo;        /* PC-offset of CALLING routine */
    unsigned prevdp;         /* data pointer */
    unsigned udescr0;        /* first half of unwind descriptor */
    unsigned udescr1;        /* second half of unwind descriptor */
    unsigned ustart;         /* start of the unwind region */
    unsigned uend;           /* end of the unwind region */
    unsigned uw_index;       /* index into the unwind table */
    unsigned prev_r19;       /* linkage-table pointer (gr19) - for PIC code */
} pframe_info;

#endif /* UNWIND_HEADER_FILE */

=====
trace.c
=====

#include <stdio.h>
#include "unwind.h"

static void copy_prev_to_curr (cframe_info *curr_frame,
                              pframe_info *prev_frame);

void unwind_trace (char *stack_trace)
{
    cframe_info curr_frame;
    pframe_info prev_frame;
    unsigned stack_ptr, space_reg, offset_reg;
    unsigned data_ptr, linkage_ptr;
    unsigned sp20, depth;
    unsigned status;

    /* set up a valid curr_frame by calling an assembly routine.
     * This assembly routine is not exported by HP, but can
     * be extracted from /usr/lib/libcl.a ... it is called

```

```
* ugetfram.o. The U_get_frame_info routine MUST be put into
* the same image as this routine. It can then set up a dummy
* curr_frame that has the correct values set.
*/
U_get_frame_info (&curr_frame);

/* U_get_frame_info doesn't zero sr0 and sr4 ... so do them explicitly
*/
curr_frame.topsr0 = 0;
curr_frame.topsr4 = 0;

/* throw away the first frame ... since its a dummy frame
* created by the call to U_get_frame_info.
*/
status = U_get_previous_frame (&curr_frame, &prev_frame);

/* Check to make sure everything is okay */
if (status)
{
    fprintf(stderr, "Stack_Trace: error while unwinding stack\n");
    return;
}

/* copy the prev_frame to the curr_frame */
copy_prev_to_curr (&curr_frame, &prev_frame);

/* Now for the real work. Initialize the trace string, and then
* loop, unwinding a frame at a time until there are no more frames
* to unwind (i.e. the offset portion of the return address is 0).
*/
*stack_trace=0;

for (depth = 0; curr_frame.currlo; depth++)
{
    status = U_get_previous_frame (&curr_frame, &prev_frame);

    /* Check to make sure everything is okay */
    if (status)
    {
        fprintf(stderr, "Stack_Trace: error while unwinding stack\n");
        return;
    }

    /* Now, we'd like to print out the return pointer. However,
    * U_get_previous_frame returns the prev_frame for the 1st NON-STUB
    * frame in the call chain. It may be the case that the return
    * pointer for this frame points into another stub. What we'd
    * really like to see is the return point for all NON-STUBS.
    * U_get_previous_frame updates curr_frame so that it contains
    * a frame whose return point is a NON-STUB. Print out this value
    * before copying over prev_frame into curr_frame.
    */
    sprintf(stack_trace + strlen(stack_trace),
        " (%2d) 0x%x\n", depth, (curr_frame.currlo & ~3));

    copy_prev_to_curr (&curr_frame, &prev_frame);
}

static void copy_prev_to_curr (cframe_info *curr_frame,
```

```

        pframe_info *prev_frame)
{
    /* Update curr_frame with values returned in prev_frame */
    curr_frame->cur_fsize = prev_frame->prev_fsize;
    curr_frame->cur_sp = prev_frame->prev_sp;
    curr_frame->cur_rls = prev_frame->prev_rls;
    curr_frame->cur_rlo = prev_frame->prev_rlo;
    curr_frame->cur_dp = prev_frame->prev_dp;

    /* don't update curr_frame.cur_r19 because U_get_previous_frame does
     * it directly.
     */
}

=====
test_shl.c
=====

#include <stdio.h>
#include <sys/signal.h>

/* This file is built into a shared library.  Thus, any traces we
 * do in here should show that our trace routine does in fact
 * work across a shared library.
 */

/* This routine shows that our trace works across an interrupt */
void sig_hand (int sig, int subcode)
{
    char trace_string[1024];

    unwind_trace(trace_string);

    fprintf(stderr, "\n\nIn sig_hand, our trace gives: %s\n",
trace_string);
    fprintf(stderr, "and U_STACK_TRACE gives: %s\n");
    U_STACK_TRACE();
    exit(0);
}

/* This routine shows that our trace works when in a shared library */
void foobar (void (*funcptr)(void))
{
    int x;
    char trace_string[1024];

    signal(SIGBUS, sig_hand);

    unwind_trace(trace_string);
    fprintf(stderr, "\n\nIn foobar, our trace gives: %s\n",
trace_string);
    fprintf(stderr, "and U_STACK_TRACE gives: %s\n");
    U_STACK_TRACE();

    /* Call back out to an archived function */
    funcptr();

    /* Force a SIGBUS ... to test trace across interrupts */
    x = *(int *)0x0001;
}

```

```
=====
test_unwind.c
=====
#include <stdio.h>

/* This file is built into an archived executable.  Traces in here
 * should prove that our trace mechanism works with archived routines.
 */

/* A prototype for a function that is in a shared library */
void foobar (void (* funcptr) (void));

/* This routine shows that our trace works when a shared library
 * routine calls back into the archived executable (2 levels deep)
 */
void foofoo (void)
{
    char trace_string[1025];

    unwind_trace (trace_string);

    fprintf(stderr, "\n\n\nIn foofoo, our trace gives: %s\n",
            trace_string);
    fprintf(stderr, "and U_STACK_TRACE gives: %n");
    U_STACK_TRACE();
}

/* This routine shows that our trace works when a shared library
 * routine calls back into the archived executable (1 levels deep)
 */
void barfoo (void)
{
    char trace_string[1025];

    unwind_trace (trace_string);

    fprintf(stderr, "\n\n\nIn barfoo, our trace gives: %s\n",
            trace_string);
    fprintf(stderr, "and U_STACK_TRACE gives: %n");
    U_STACK_TRACE();

    foofoo();
}

/* This routine shows that our trace works for archived functions */
void bar (void)
{
    char trace_string[1025];

    unwind_trace (trace_string);

    fprintf(stderr, "\n\n\nIn bar, our trace gives: %s\n", trace_string);
    fprintf(stderr, "and U_STACK_TRACE gives: %n");
    U_STACK_TRACE();

    foobar(barfoo);
}

void foo (void)
```

```

{
    bar();
}

main()
{
    foo();
}

=====
Ugetfram.s
=====

.CODE
;-----
;
;                               U_get_frame_info
;
; U_get_frame_info loads the value of the caller's SP, PCspace, PCOffset
;
; and DP into a record, a pointer to which has been passed into
; this routine in arg0. The format of this record is that required by the
; unwind routine :
; "U_get_previous_frame".
;
;
; offset contents
;
; 0      cur_fsize      framesize of called routine
; 4      curSP          SP of called routine
; 8      curRLS         PC_space of calling routine
; 12     curRLO         PC_offset of calling
;                          routine :
; 16     curDP          DP of called routine
; 20     topRP          RP (reg. 2) of called routine
;
; 24     topMRP         MRP (reg. 31) of called routine
;
; 28     cuffSR0
; 32     cuffSR4
; 36     curR3
; 40     cur_r19(new offset)
;endif
;
; INPUT PARAMETERS:
;   arg0 : pointer to a 11-word structure with the above
;         layout :
;
; OUTPUT PARAMETERS:
;   the fields curSP, curRLS, curRLO, curDP
;
;-----
;
U_get_frame_info
    .PROC
    .CALLINFO
    .ENTRY
    stw    sp,4(arg0)        ; store caller's SP
    mfsp   sr4,r20
    stw    r20,8(arg0)       ; store caller's PC space

```

```
        stw      rp,12(arg0)      ; store caller's PC offset
        stw      dp,16(arg0)      ; store caller's DP
        stw      r3,36(arg0)      ; store caller's R3
        stw      r0,0(arg0)       ; initialize rest of fields
        stw      r0,20(arg0)      ;   -"-
        stw      r19,40(arg0)     ; fetch r19
        bv       r0(rp)           ; return, after restoring SP
        .EXIT
        stw      r0,24(arg0)      ;   -"-
        .PROCEND

        .EXPORT U_get_frame_info, CODE, PRIV_LEV=3
        .END
```

```
=====
output.txt
=====
```

In bar, our trace gives:

```
( 0) 0x2044
( 1) 0x20c0
( 2) 0x20f8
( 3) 0x800419a4
( 4) 0x18fc
```

and U_STACK_TRACE gives:

```
( 0) 0x0000208c   bar + 0x60  [./example.out]
( 1) 0x000020c0   foo + 0x14  [./example.out]
( 2) 0x000020f8   main + 0x14  [./example.out]
( 3) 0x800419a4   start + 0x70  [/lib/libc.sl]
( 4) 0x000018fc   $START$ + 0x9c  [./example.out]
```

In foobar, our trace gives:

```
( 0) 0x8084844c
( 1) 0x20a0
( 2) 0x20c0
( 3) 0x20f8
( 4) 0x800419a4
( 5) 0x18fc
```

and U_STACK_TRACE gives:

```
( 0) 0x808484d0   foobar + 0xe0  [/tmp/unwind_example/example.sl]
( 1) 0x000020a0   bar + 0x74  [./example.out]
( 2) 0x000020c0   foo + 0x14  [./example.out]
( 3) 0x000020f8   main + 0x14  [./example.out]
( 4) 0x800419a4   start + 0x70  [/lib/libc.sl]
( 5) 0x000018fc   $START$ + 0x9c  [./example.out]
```

In barfoo, our trace gives:

```
( 0) 0x1fb4
( 1) 0x808484f4
( 2) 0x20a0
( 3) 0x20c0
( 4) 0x20f8
( 5) 0x800419a4
```

(6) 0x18fc

and U_STACK_TRACE gives:

```
( 0) 0x00001ffc  barfoo + 0x60  [./example.out]
( 1) 0x808484f4  foobar + 0x104  [/tmp/unwind_example/example.sl]
( 2) 0x000020a0  bar + 0x74   [./example.out]
( 3) 0x000020c0  foo + 0x14   [./example.out]
( 4) 0x000020f8  main + 0x14  [./example.out]
( 5) 0x800419a4  start + 0x70  [/lib/libc.sl]
( 6) 0x000018fc  $START$ + 0x9c [./example.out]
```

In foofoo, our trace gives:

```
( 0) 0x1f30
( 1) 0x2008
( 2) 0x808484f4
( 3) 0x20a0
( 4) 0x20c0
( 5) 0x20f8
( 6) 0x800419a4
( 7) 0x18fc
```

and U_STACK_TRACE gives:

```
( 0) 0x00001f78  foofoo + 0x60  [./example.out]
( 1) 0x00002008  barfoo + 0x6c   [./example.out]
( 2) 0x808484f4  foobar + 0x104  [/tmp/unwind_example/example.sl]
( 3) 0x000020a0  bar + 0x74   [./example.out]
( 4) 0x000020c0  foo + 0x14   [./example.out]
( 5) 0x000020f8  main + 0x14   [./example.out]
( 6) 0x800419a4  start + 0x70  [/lib/libc.sl]
( 7) 0x000018fc  $START$ + 0x9c [./example.out]
```

In sig_hand, our trace gives:

```
( 0) 0x8084830c
( 1) 0x800ab3e8
( 2) 0x808484fc
( 3) 0x20a0
( 4) 0x20c0
( 5) 0x20f8
( 6) 0x800419a4
( 7) 0x18fc
```

and U_STACK_TRACE gives:

```
( 0) 0x8084838c  sig_hand + 0xb4  [/tmp/unwind_example/example.sl]
( 1) 0x800ab3e8  sigreturn [/lib/libc.sl]
( 2) 0x808484fc  foobar + 0x10c  [/tmp/unwind_example/example.sl]
( 3) 0x000020a0  bar + 0x74   [./example.out]
( 4) 0x000020c0  foo + 0x14   [./example.out]
( 5) 0x000020f8  main + 0x14   [./example.out]
( 6) 0x800419a4  start + 0x70  [/lib/libc.sl]
( 7) 0x000018fc  $START$ + 0x9c [./example.out]
```

8.5 Setjmp and longjmp jmp_buf

Setjmp and longjmp functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program. Setjmp saves its stack environment in *env* (jmp_buf type) for later use by longjmp. Longjmp restores the environment saved by the last call of setjmp with the corresponding *env* argument. After longjmp is completed, program execution continues as if the corresponding call of setjmp had just returned the value *val*.

```
#include <setjmp.h>
```

```
int setjmp (env)
```

```
jmp_buf env;
```

```
void longjmp (env, val)
```

```
jmp_buf env;
```

```
int val;
```

```
struct jump_buffer {
```

```
int jb_rp;                /* Return Pointer */
```

```
int jb_sp;                /* Marker SP */
```

```
int jb_sm;                /* Signal Mask */
```

```
int jb_os;                /* On Sigstack */
```

```
int jb_gr3;               /* Entry Save General Registers */
```

```
int jb_gr4;
```

```
int jb_gr5;
```

```
int jb_gr6;
```

```
int jb_gr7;
```

```
int jb_gr8;
```

```
int jb_gr9;
```

```
int jb_gr10;

int jb_gr11;

int jb_gr12;

int jb_gr13;

int jb_gr14;

int jb_gr15;

int jb_gr16;

int jb_gr17;

int jb_gr18;

int jb_gr19;

int jb_sr3;          /* Entry Save Space Register */

double jb_fr12;      /* Entry Save Floating Point Registers */

double jb_fr13;

double jb_fr14;

double jb_fr15;

int jb_sm2;          /* Second word for Signal Mask */

double jb_fr16;

double jb_fr17;

double jb_fr18;

double jb_fr19;

double jb_fr20;

double jb_fr21;

int jb_rp_prime; /* rp prime from frame marker */

int jb_ext_dp; /* external_dp from frame marker */

};
```

```
typedef struct jump_buffer jmp_buf;
```

8.6 Process Context

This section discusses exception handling in Ada and C++.

8.6.1 Ada Exception handling

The exception handling semantics of the Ada/800 runtime are implemented in the package `EXCEPTION_MANAGER`, which is nested in the package `ADA_RUNTIME`, and its subunits.

The code generator generates instructions to raise exceptions along with tables that describe the actions to be taken when an exception is raised. There are two methods to enter `RAISE_EXCEPTION`, the procedure that handles exceptions. One is through system traps and the signal handler of the runtime system, the other one is by invoking `RAISE_EXCEPTION` directly. The trap instructions are used for checks. When a check fails, it will trap, and the signal catcher of the runtime system will receive a HP-UX signal. It then decodes the trap and calls `EXCEPTION_MANAGER.RAISE_EXCEPTION` to treat the exception. For explicit raise statements, the code generator will emit direct calls to `RAISE_EXCEPTION`. The `RAISE_EXCEPTION` routine can also be called by other routines in the runtime system.

`RAISE_EXCEPTION` takes two pieces of information from its caller: the exception code and the program counter where the exception occurred.

Exception Through Traps and Signals

The operation to handle signals is as follows:

- exception occurs
- system trap occurs
- signal generated
- signal handler entered
- `RAISE_EXCEPTION` called

The exception manager looks up the exception in a set of tables, depending on the place where the exception occurred and what exception it was, and tries to find a handler. If a handler is found, the exception manager does all cleanups necessary (all cleanup actions in scopes from the scope that raised the error down to, but not including the destination scope) and then passes control to the handler. The necessary cleanups include waiting for dependent tasks, freeing heap memory, and cutting back the stack to reclaim the space for dynamic objects. If no handler is found, the exception is propagated.

Exception Tables

The object code contains scope and handler tables which are searched by the exception manager. The linker emits the scope table from information in the the relocatable object (SOM) files. The code generator emits the handler table directly.

There is a scope table created by the linker for a program. The scope table consists of scope entries. There are two symbols (beginning and ending symbols) provided by the linker to delimit the scope entries. The beginning and ending symbols point to the first entry and the entry past the last one (first byte not in the table) respectively. The structures of a scope table and a scope table entry are as follows:

Scope Table
entry #1
entry #2
...

Scope Table Entry
scope_begin
scope_end
handler_tbl_addr

The *scope_begin* and *scope_end* are the beginning and ending offsets associated with the current scope. The *handler_tbl_addr* is the code address of the handler table for this scope.

The scope entries need to be sorted in the order of the code generator encountering the end of scope. This means that in the table, the "scope_end" values will be monotonically increasing. The linker has to be changed to allow stacked scopes, i.e., an inner scope is totally nested within an outer scope.

We only need a scope entry for the statement region of a scope that has a handler. This is a great simplification from previous schemes. We do not need scope entries for separate package bodies, since unwind will be able to unwind through a separate package body and find the invocation location.

When an exception is raised in a block which is nested in another block, if the handler of the inner scope doesn't handle it, it will be propagated to the outer scope. It means that the "scope_begin" and "scope_end" of the outer scope has to enclose those of the inner scope. The current UCODE implementation unnests scope entries such that entries do not have overlapping "scope_begin" and "scope_end". An Ada option will be added to the UCODE such that entries will not be unnested. It also has the advantage of having less scope entries than entries which are unnested.

The exception parts are placed in line, to follow immediately after the statement part of the scopes. While this introduces a branch over the exception part in the normal case, it makes the tables simpler, because a table entry is not needed to know how to propagate out of the exception scope. This decision was also necessary due to the requirements imposed by existing HP-PA utilities that demand that a procedure have only one exit, and this exit must be the last instruction in the procedure. This requirement precludes putting exception parts in the "dead space" after the procedure exit.

There is a one-to-one correspondence between scope table entries and handler tables. This means that the handler table is basically an extension of the information in the scope table entry. It would be possible to merge the two tables, and eliminate the word necessary for the "handler_tbl_addr". For now, the tables are kept separate, to be consistent with Pascal's method of handling try/recover information, which uses a similar 3 word scheme to describe try/recover scopes. Pascal does not need the handler table information, so their 3rd word is simply the code address of the recover part. In order to stop Pascal from executing code in the handler table, a special trap instruction is needed at the beginning of the handler table and its trap handler will stop the program. The special trap instruction is necessary because the handler table is not in the unwind region. The handler table is put in the code space and is right after the code for a procedure. As unwind regions only cover code and the handler table is between the code for different procedures, the handler table is not in the unwind region.

A handler table consists of a header and an array of handler entries. The code address in the scope table entry points to the handler table header. This header describes the scope which has an exception part. Following this header are entries which describe each individual exception which has a handler, and the code address of that handler.

The structures of a handler table, a handler table header and a handler entry are as follows:

Handler Table
handler_table_header
handler_table_entry #1
handler_table_entry #2
...

Handler Table Header
trap_instruction
number_of_entries
scope_kind
anonymous_raise_save_offset

Handler Table Header
sp_save_location
cleanup_stop_point

Handler Table Entry
exception_code
handler_code_addr

The *number_of_entries* field simply denotes how many handler table entries follow this header.

The *scope_kind* denotes the kind of scope we have, such as procedure, function, package, accept body, etc. This is necessary in the case where we have an exception occurring in the statements covered by these handlers, but the exception we have is not handled here. We therefore need to propagate, and the propagation is different depending on the *scope_kind*. If the scope is an inline package or block, we simply need to look for a handler in the scope tables immediately following the current scope table entry. We continue this linear search till we either find an enclosing entry or we get to a subprogram type entry which starts after the exception raising offset. The *scope_kind* field is necessary to terminate this search. If the scope is a subprogram or separate package body, we need to call unwind to determine the call or invocation location.

The *anonymous_raise_save_offset* is an offset to a frame location used to save the current exception code before giving control to the handler. It is also used to restore the exception code when any anonymous raise is executed. It has the value of undefined offset unless one of the handlers contains an anonymous raise.

The *sp_save_location* is an offset to a frame location used to restore the stack pointer before transferring control to one of the handlers. The object code stored the stack pointer into this frame location at entry to the statement region covered by this exception part. It has the value of an invalid frame offset if the unit does not contain further blocks which have dynamic variables. If it is the invalid frame offset, no stack cut back is necessary. The *cleanup_stop_point* is an offset to a frame location that describes where to stop the process of tasking and heap cleanups. Before transferring control to one of the handlers, we need to wait for all dependent tasks on the chain above *cleanup_stop_point*, and after that, free all heap objects on the chain above *cleanup_stop_point*. The code generator will set *cleanup_stop_point* to be the top of the logical frame for the block that has the handler. This will cause cleanup of all objects on the chain above those of this particular block (and NOT including this block). Note that it doesn't matter if the block has cleanups or not, since we have only a top of frame point, and not the location of the cleanup list entries for the block. All the offsets in the handler table header (*anonymous_raise_save_offset*, *sp_save_location*, and *cleanup_stop_point*) are always offsets from PSP even on static frames.

The *exception_code* is the value of an exception handled here, or a special value to rep-

resent a "when others". The *handler_code_addr* is a subprogram symbol plus an offset which points to the handler code area for the particular exception code. As the handler table is put in the code space and is right after the end of a procedure, *handler_code_addr* is relocation fixup. In the case of nested blocks, the handler code is inline but the handler table is moved out of the line (to right after the end of a procedure). The reason for this is that the unwind regions delimit the "begin" and "end" of code which should not have any data, otherwise, the linker may do undesired fixups. Here is an example of code layout :

```
begin
    statements #1
    begin
        statements #2
    exception
        handlers # 2
    end;
exception
    handlers #1
end;
```

The layout of generated code and table is as follow:

```
code for statements #1
code for statements #2
code for handlers #2
code for handlers #1
handler table for the inner scope
handler table for the outer scope
```

When `RAISE_EXCEPTION` is called, it calls various routines to perform the following operations:

1. `FIND_SCOPE_ENTRY` routine uses two steps to locate the desired scope entry. It first does a binary search to find the first entry that ends on or after the raising offset, and then a linear search from there to find the first entry including the raising offset. This algorithm works because the entries are emitted in order of the code generator encountering the end-of-scopes. This means that entries from inner blocks will come before entries from outer blocks.
2. If no scope entry is found, then there is no local handler within the current scope, and we need to propagate by calling `unwind` to go back one scope, and then repeat step 1.
3. If the scope entry is found, `FIND_HANDLER_TABLE` routine finds the handler table based on the handler table offset in the scope entry.
4. The `FIND_HANDLER` routine loops through the handler table to search for the handler entry whose exception code matches with the exception raised.

5. If a handler is found, save the current exception at the handler table's `anonymous_raise_save_offset` if necessary, do any necessary cleanups, and pass control to the handler; otherwise, propagate the exception. If we need to propagate, and the scope kind is either a procedure or a separate package body, then call unwind to go back one scope, and then start this entire search process again. If we need to propagate from a block or inline package, simply keep searching forward in the scope table starting with the next entry after the current one. In this way, if there is an enclosing block or procedure with a handler, we will find it based on the original offset.

Unwind Mechanism

The exception handling routines need to call lower level utilities to unwind through a scope. The existing unwind mechanism for Spectrum is used for this purpose, with some extensions to allow handling of variable sized scopes and separate packages. The primary routine provided by the unwind utilities is the `u_get_previous_frame` routine. Given a PC value, this routine gets the appropriate unwind entry for this scope from the unwind table. If a scope is a subprogram, then its previous scope is its call; if a scope is a separate package body, then its previous scope is its invocation point. The information kept and updated by the unwind routines reflects the program state in that scope. This information consists of the SP, DP, PC space, PC offset and the size of the current scope. When running under HP-UX, DP and the PC space values do not change.

The unwind utilities find out the scope size of any scope by looking at the unwind descriptor for that scope. In Ada, the scope size is not known at compile time for procedures that either allocate dynamic objects or have separate packages. Thus, the scope size cannot be looked up in the unwind descriptor. A boolean flag will be set in the unwind descriptor to indicate a variable sized scope. For such procedures, the previous stack pointer (PSP) will be stored in the frame marker upon entry to a procedure. This stored PSP will be used to get the SP for the previous (caller's) frame.

When a procedure allocates a dynamic object, the size of the frame is increased at that point. The frame marker is also moved out. This extension of the frame is done by a special millicode routine. Within this routine the SP changes in value. The unwind descriptor of the frame extension millicode routine is specially marked, so that the unwind utilities can recognize the fact that SP changes value within the routine and can unwind appropriately. This implies that the unwind utilities have to know about the sequence of instructions in the millicode routine completely.

On exit from a block that allocates dynamic objects, the frame has to be shrunk to the correct size. This is again done through a special millicode routine. However, no special treatment is needed to unwind through this frame size reduction millicode. The SP and the frame marker are in a valid state at each instruction in the routine.

The elaboration code of a separate package is in the same frame as the parent unit. The entry and exit sequences of this code are different from ordinary procedures. The unwind descriptor for a separate package elaboration has to indicate that it is part of a parent frame. However, the base of the frame contribution of the package must be stored

in the descriptor so that the spill locations for the separate package can be identified.

There is one thing we have to be extremely careful about while propagating exceptions. The meaning of offsets into the code may be different depending on the situation. If we are starting to unwind from a trap, we will have the actual trapping offset. If we are unwinding through a procedure call, the return point is two instructions past the call branch. The reason it is two past and not one past is that the instruction immediately following the call branch is executed before the call actually occurs, in the delay slot of the branch.

Cleanups for Dependent Tasks, Heap Objects, and Stack Space

There are three separate cleanups necessary before transferring control to any handler. These are for tasks, heap objects, and stack space used by dynamic variables. For tasks and heaps, there are two chains that contain cleanup information. The cleanups necessary are for entries on these lists above the frame contribution for the scope of the destination exception handler. This assumes that frame contributions are sorted, in that a nested scope's frame contribution will always be above the contributions of scopes that it is nested within. This scheme is described in detail in another memo.

To cut the stack back to reclaim space used for dynamic variables, use `sp_save_location` field as the offset to a frame location containing the value of SP that we need to restore. This is necessary only for scopes that contain blocks with dynamic variables. If `sp_save_location` contains the `invalid_offset` then we do not need to restore SP. Note that we must change SP and then copy the frame marker back to correspond to the new SP value. This means that we need to modify the value of SP in our saved register image and also move the stack marker. Note that this cleanup scheme only handles cleaning up dynamic variables in the frame of the handler. This works since the unwind procedure will get rid of dynamic objects in frames above the destination, since their dynamic objects will disappear when their frame disappears.

Note that we must do the tasking cleanups before the heap cleanups or stack reclamation, since the tasks may be using the heap objects or dynamic variables.

Procedure Traceback Tables and Mechanism

When an exception is raised but not handled, we want to give a stack traceback, which provides information about the path of exception propagation through active scopes. The traceback routine prints the exception name, followed by the stack trace which prints the compilation unit name(s), scope name(s), scope kind(s), and line number(s) of each scope it encounters. This information all comes from the unwind and auxiliary unwind tables (generated by the linker), the name tables (generated for each compilation unit by code generator), and the exception name table (generated for the entire program by the binder). For traceback, we do not use information from the exception tables themselves. The scope part is not printed, since it would take more space in the auxiliary unwind table to support it, and we are unsure that the user really wants to see it. Also, we decided to suppress the tracing of blocks and local packages, so there will be no entries in the unwind tables for them. There will, however, be entries for separate packages, so the user will see trace propagation through separate packages.

When the exception manager goes all the way to the base of the stack and cannot find any applicable handler, then it knows that the current task will terminate. At this point, we will print a traceback. The exception manager will call the traceback routine, passing the original raising offset.

The linker generates the unwind and auxiliary unwind tables, and these two tables are parallel. For each entry in the unwind table, there is a corresponding entry in the auxiliary unwind table. Their structures are as follows:

Unwind Table
unwind entry #1
unwind entry #2
...

Unwind entry
scope_begin
scope_end
flags
frame_size

aux_unwind_table
aux unwind entry #1
aux unwind entry #2
...

aux unwind entry
cu_name_ptr
scope_name_ptr
scope_kind
line_number_tbl_ptr

The unwind table is similar to the scope table for exception handling, except that the unwind table has more entries and entries are unnested. It has an entry for each subprogram level scope as well as library level packages and separate packages. There are no entries for local packages as it would require multiple unwind entries for a subprogram level scope. In other words, we forgo having traceback for any scopes smaller than subprograms (ie: blocks and local packages). The *scope_begin* and *scope_end* have the same meaning as the exception tables, and are sorted in the same order. The *flags* can include a boolean indicating the presence of a variable sized frame.

Both *cu_name_ptr* and *scope_name_ptr* point to the name table which contains the string literals emitted by code generator. A null pointer for *cu_name_ptr* or

scope_name_ptr indicates "unknown" which may occur when units are compiled with traceback off. If the value of scope_name_ptr is 1, it indicates that it is a "<type support subprogram>". The *line_number_tbl_ptr* points to the line number table generated by the linker. The linker uses the information from the start of statement fixups to emit the line number table. The code generator provides cu_name_ptr, scope_name_ptr and scope_kind to let the linker emit the auxiliary unwind table. Ada will use an option to pass these to Ucode, and there is a new fixup called R_AUX_UNWIND that Ucode will use to communicate these items to the linker.

The traceback routine calls **u_get_previous_frame** which takes an offset to get the appropriate unwind entry from the unwind table. Once the unwind entry is located, the corresponding entry in the auxiliary unwind table can be located easily as both tables consist of parallel arrays of entries. A unwind utility routine will return either an index or offset from the beginning of the unwind table such that the runtime routine can find the associated auxiliary unwind entry.

The name table generated by the code generator consists of compilation unit names and scope names pointed by cu_name_ptr and scope_name_ptr in the auxiliary unwind table. The string literals in the name table end with a null character. There is one name table for each compilation unit (not just a single one for the entire program). There is no need to ever traverse the name tables, so we do not need pointers or symbols to the start of them. The only action supported by these tables is direct access to a name pointed to by an entry in the auxiliary unwind table.

The structure of a name table is as follows:

Name Table of a cu
cu_name
scope_names

The exception names table has a different format, however. We need to be able to search this table, looking for an entry with a given exception value. We do not have direct pointers into entries in this table. The binder will write out a single exception names table for the entire program, using information provided by the code generator. Note that if we are suppressing traceback, we do not need to produce either the exception names table or the names tables.

The structure of the exception names table is as follows:

Exception names table
exception name entry #1
exception name entry #2
...

Exception names entry
exception_value
exception_name_length
exception_name_text

The exception names table is thus a table containing entries of varying lengths. Each entry starts with an exception value, and then supplies the length and the characters of the name of that exception. All exceptions will have entries in this table, including the predefined and I/O exceptions.

The structure of line_number_table is as follows:

Line Number Table			
version_number			
start_code_offset			
start_line_number			
code_diff	line_diff	code_diff	line_diff
code_diff	line_diff	code_diff	line_diff
...

The line_number_table starts with a header that contains three words of information: version_number, start_code_offset and start_line_number. The first word contains the version number, as a 32-bit integer. It is initially 1 (one), and is here to support the possibility of changing the table format sometime in the future. If we ever change the format, the runtime would have to look at the version number, and interpret the tables based on the format that corresponded to that version. This would allow the intermixing of old format and new format SOMs into one executable.

The second word gives the offset (in BYTES) from the beginning of the unwind region to the start of the code for the first numbered line in the region. The third word gives the line number of that first line of code. After that, the line_number_table contains any number of elements, each of which has a pair of code_diff and line_diff which are usually the differences of code offsets and line numbers between the previous and current elements. The code_diff values are word offsets (not byte offsets), and are unsigned (range 0 .. 255). The line_diff values are signed, range (-128 .. 127). Both entries in this pair (code_diff and line_diff) occupy only one byte each. For entries with too large a span to fit into one byte fields, the table simply uses additional entries, in the standard format. Note that since the first line of code is covered by the information in the two-word header, the first pair of (code_diff, line_diff) gives the information for the second line of code.

The meaning of each pair in the table is determined by the value of `code_diff`, as shown in the table below.

Code_diff_value	Meaning
0 to 250	Normal entry, representing differences.
251	The next 3 bytes (the <code>line_diff</code> , and the following <code>code_diff</code> and <code>line_diff</code>) contain a secondary line number.
252	This is the short form of a secondary line number. The following (signed) <code>line_diff</code> gives the difference between the last secondary line number and the value of the current one.
253	The next 3 bytes (the <code>line_diff</code> , and the following <code>code_diff</code> and <code>line_diff</code>) contain an absolute line number. This is used in place of many difference entries, when the difference between two line numbers is large. This must follow an entry with a <code>line_diff</code> of zero (see below).
254	Ignore this pair and the following two pairs (the 254 and the following <code>line_diff</code> , <code>code_diff</code> , <code>line_diff</code> , <code>code_diff</code> , and <code>line_diff</code>). This is for future expansion.
255	End of the table. Stop searching. On this entry, the value of <code>line_diff</code> is meaningless

The special code value of 251 means that a secondary line number follows in the next three bytes. These entries give an extra line number for lines that are the result of instantiations or inlinings. In this case, the regular table entries give the line number of the inlining or instantiation, and the secondary line number gives the original line number of the source (before it was inlined or instantiated).

The special code value of 252 is the short form of the above. Instead of taking three bytes to give the absolute value of a secondary line number, this form gives one byte which when added to the value of the last secondary line number, gives the value of this secondary line number. The following byte is signed, so the secondary line number can go either up or down. The following byte can be zero, which means this secondary line number is exactly the same as the last one. Note that the previous secondary line number (that we base the value on) can be either a short form or a long form).

The special code value of 253 is an absolute line number. When encountering one of these, we disregard our line number value we have built using the `line_diffs`, and use this new value. From then on, new `line_diffs` apply to the new value. Note that since this form does not have a `code_diff` field, it is equivalent to a (`code_diff`, `line_diff`) pair with a `code_diff` value of zero. Entries like this have the effect of hiding the line that comes before it. To prevent this, before the code 253 special entry, we need a regular entry with a zero value for `line_diff`, and a `code_diff` value giving the length of the previous line (see example).

Because a code 253 entry takes up 6 extra bytes, we should only use it when we would otherwise have to use more than 4 regular entries to span a large gap in line numbers.

During the traversal of the table, we do not use the secondary line number entries (with codes of 251 or 252) as we attempt to find the line number for a given code offset. After we find the line number, we look at the following entry, to see if it is a secondary line number entry. If so, it applies to the line we just found. If not, there is no secondary line number for that line, which means the line was not involved in inlining or generic instantiation.

This line number table, and the associated auxiliary unwind tables are now a CLL standard, and will be produced by the linker. So far, Ada is the only language to use the secondary line numbers, and no language uses the special code 254. Code 253 can be used by any language, but BASIC is the language that will probably use it the most. The design of the tables is such that when there are no secondary line numbers (such as for the other languages), there is very little overhead to support them.

It is likely that any future use of the code 254 would have to change the number of bytes to skip. This is OK, since if we made that change we would also change the version number, so the runtime could use the new method.

The best way to demonstrate this method is by example. Suppose you have some code with the following characteristics (all starting positions and sizes in the following table are in WORDS):

Line #	Secondary Line #	Start at:	Size
1	-	0	10
2	-	10	20
3		30	400
4	-	430	10
200	-	440	10
205	-	450	10
180	-	460	300
30	4000	760	10
30	4001	770	20
30	4005	790	15
3000	-	805	22
3010	-	827	25

Note that the above example has line numbers which are not monotonically increasing. The line numbers are also not very regular. There is some code at lines 1, 2, 3, and 4, and then a lot of comments, so the next line of code is at line 200. Furthermore, after line 205, we have some unnested exception handlers, with lines at 180 and then at 30.

Finally, we have lots of comments, followed some final lines at line numbers 3000 and 3010. Line 3 is an inlining of a single-line subprogram (originally at line 95), and line 30 is an instantiation of a 3 line generic (originally at line 4000 to 4005). The code offsets are monotonically increasing, as you would expect.

Here is the line number table that would represent the above situation. The picture below shows the actual table content in the first two columns only. The last columns indicate the meaning of that entry.

start_code_offset	start_line_number	Means	Actual Offset	Line #
0	1	= >	0	1

code_diff	line_diff	Means	Actual Offset	Line #
10	1	= >	10	2
20	1	= >	30	3

code_diff	extra info	Means	secondary line number
251	0, 0, 95	= >	95

code_diff	line_diff	Means	Actual Offset	Line #
250	0	= >	280	3

code_diff	extra info	Means	secondary line number
252	0	= >	95

code_diff	line_diff	Means	Actual Offset	Line #
150	1	= >	430	4
10	127	= >	440	131
0	69	= >	440	200
10	5	= >	450	205
10	-25	= >	460	180
250	0	= >	710	180
50	-128	= >	760	52
0	-22	= >	760	30

code_diff	extra info	Means	secondary line number
251	0, 15, 160	= >	4000

code_diff	line_diff	Means	Actual Offset	Line #
10	0	= >	770	30

code_diff	extra info	Means	secondary line number	
252	1	= >	4001	

code_diff	line_diff	Means	Actual Offset	Line #
20	0	= >	790	30

code_diff	extra info	Means	secondary line number	
252	4	= >	4005	

code_diff	line_diff	Means	Actual Offset	Line #
15	0	= >	805	30

code_diff	extra info	Means	Actual Offset	Line #
253	0, 11, 184	= >	805	3000

code_diff	line_diff	Means	Actual Offset	Line #
22	10	= >	827	3010
255	0	= >	stop	stop

The algorithm for traversing the tables is as follows. You start with the offset of code that was active at the time of the traceback. The goal is to find the line number whose code contains this offset. As you traverse the table, you start with `Start_code_offset` and `Start_line_number`. When you encounter a regular table entry pair (one with a `code_diff` in the range 0 .. 250), add its code offset difference to your code offset counter, and its line number difference to your line number counter. When you encounter a code 253 entry, update your line number from the next three bytes in the table. The last two columns of the table above shows the value of these two counters after encountering each such entry in the table.

The algorithm stops when it finds the last actual code offset (the one furthest along in the table) that is less than or equal to the offset you are looking for. In practical terms, this means find the first actual code offset bigger than the one you are looking for, and then back up one entry in the table.

Let's look at a few examples:

Example 1. If you are looking for offset 90, you find line 3, since 30 is the last offset \leq 90. You then notice that this piece of line 3 has a (long form) secondary line number of 95.

Example 2. If you are looking for offset 300, you also find line 3, but you find its second entry, since 280 is the last offset \leq 300. Here also, this piece of line 3 has a (short form) secondary line number of 95.

Example 3. If you are looking for offset 440, you find line 200, since its offset of 440 is the last offset \leq 440. Note that the entry for line 131 (there really is no line 131) is not

reachable, since its actual offset of 440 is the same as the next entry's actual offset. Note that there is no secondary line number on this line.

Example 4. If you are looking for offset 805, you find line 3000, since its offset of 805 is the last offset ≤ 805 . Once again, there is a hidden line before it, also with an offset of 805, which is just necessary to establish the code offset for the absolute line entry for line 3000. Since line 3000 shares the same code offset as the last entry for line 3, that last entry for line 3 is not reachable. This unreachable entry must exist, to establish the starting offset for the line 3000 entry.

This method uses only two bytes per line in most cases. If the size of an individual line is bigger than 250 words, or the difference in line numbers is not in the range -128 .. 127, then you will end up with extra regular entries in the table. You may end up with many extra regular entries on unusual cases. For example, a line containing 1100 words of object code would need five regular table entries.

Any line with a secondary line number will also use more space, to hold the extra information.

Any extra regular entries inserted in the table will be of one of two types:

1. A duplicate entry for a given line number, to get around the limit of 250 words of object code per line (as shown by lines 3 and 180 in the example). Note that if the long line also has a secondary line number, we need a secondary line number entry to follow each piece of the original line in the table (as shown by line 3 in the example). Note that after the first secondary line number entry, the duplicate ones can be the short form (code 252) with a difference value of 0.
2. A hidden entry with an unused line number, to either get around the limit on the span between line numbers (as shown by lines 131 and 52 in the example), or to precede an absolute line number entry (as shown by the last entry for line 3 in the example). These entries can never show up in a traceback because they share the same actual offset with the entry that follows them. Note that in this case we do not need secondary line numbers on these hidden lines.

Note that if any line needs both classes of extra regular entries (if it has more than 250 words of code AND has a line number that differs from the next one by an amount other than -128 .. 127), then all the class 1 extra entries must precede the class 2 ones. This is necessary, since you must first take into account all the offsets for the given line number, before you start changing the line number for the next line. This is shown by line 180 in the example.

The advantages of this method are that it is very compact, in that most lines will have only one entry, occupying 2 bytes total, and that the necessary extensions are simple, and don't effect the processing speed very much.

8.6.2 C++ Exception handling

The C++ exception handling can be broken down into four functional areas:

1. Transfer of Control

When an exception is encountered, the exception handling mechanism must suspend execution at the throw point, and resume execution at the appropriate catch point. When execution is resumed, global and local variables must have correct values.

1. Exception Identification

The exception handling run-time support (henceforth simply the “run-time”) must have type information available which describes various characteristics of a type; for example, this information is used to determine if a thrown exception is handled by a catch clause. The mechanism for emitting and utilizing this information is called “Exception Identification”.

1. Object Cleanup

When an exception occurs, the exception handling mechanism should attempt to destroy all fully and partially constructed automatic objects between the throw point and the catch point. If an exception occurs in the construction of a heap object, the heap object should be destroyed and any memory allocated for the object should be deallocated. When *exit* is called, fully and partially constructed static objects should be destroyed.

1. Storage Management

The run-time must maintain a copy of a thrown object. There can be multiple thrown objects which are simultaneously active, and the run-time must manage the memory necessary to store such objects.

Implementation Scheme

The exception handling scheme can be summarized as follows:

- Use `setjmp/longjmp` to transfer control from a *throw* to the appropriate *catch* clause.
- Use a linked list of “markers” running through the stack to record the execution of try blocks, functions with exception specifications, and functions which require object cleanup.
- The translator emits *typeinfo* objects to store useful information about a type (such as the list of base classes). This information is used by the exception mechanism to determine if a catch clause can handle the thrown object, and to check for exception specification violations. The *typeinfo* information is also used to determine how to destroy partially constructed objects.
- Upon entry into a function which requires object cleanup in the event of an exception, a “cleanup marker” is chained into the chain of markers. This cleanup marker will point to a statically generated table which describes the cleanup actions required by this function.
- The chain of markers is also used to handle functions with exception specifications; this is done by adding a “specification marker” to the marker chain upon entry to a function with an exception specification.

