PRECISION RISC ORGANIZATION

19111 Pruneridge Avenue, Cupertino, California 95014-9807 U.S.A.

From: Bob Campbell **Date:** October 7, 1993

To: PRO Technology Council **Subject** PRO Application Binary Interface 1.0

Greetings:

Enclosed is the final draft of the PRO ABI. This draft is submitted for the approval of the Technology Council at the November 4th, 1993 meeting to be hosted by Winbond in Taiwan.

The following pages list the major changes from the 3rd draft, and change bars are used within the document itself.

I would like to express my thanks to everyone for the valuable comments provided. This document would not have been possible without your help.

Best regards,

Bob Campbell
Precision Risc Organization
M/S 44MU
19111 Pruneridge Avenue
Cupertino, CA 95014-9807
USA

Phone: 1-408-447-5687
Fax: 1-408-447-7568
E-mail: campbelr@cup.hp.com

Document # 1PRO-180

Chapter 1

1-7	Usage changed to FPR to match Architecture Manual.
1-9	Additional related documents added.

Chapter 2

2-6	Definition for /sbin/rc#.d corrected.

Chapter 3

3-3	BREAK instruction supported. Required by crt0 and test tools.
3-10	Alignment example modified to show larger bitfield.

Chapter 4

4-12	GR 22 noted as used for direct system calls.
4-18	Caller must ensure space is allocated for >64 bit return values.
4-27	Pseudo code for plabels clarified.
4-28	The current section for stack unwind reflects the best available documentation available. It is expected that this section will be improved in a future edition or as an errata supplement.

Chapter 5

5-8, 5-10	The entry_space and entry_subspace fields are changed to reserved fields. They are not currently used on HP-UX.
5-9	The version_id field now includes the value for relocatable objects.
5-10	entry_offset is changed to indicate the usage of the 2 low-order bits.
5-14	fixup_request_total is the number of bytes.
5-16	Implementation-specific headers are prohibited. They are not portable.
5-20	The exec_entry field incorrectly specified the usage of the low-order bits.
5-26	The optional nature of init_pointers was stressed.
5-31, 5-36	The field initializer_count was defined to support multiple initializers.
5-42	The term "absolute path" is used to reflect the usage in the standards.
5-46	The definition of bypassable was improved.
5-49	The fact that the hash value was hash_string modulo the number of hash table entries was clearly stated.
5-53	The original definition for drelocs, imports, and module_dependancies was incorrect. There is an additional level of indirection, and the relationship between imports and module dependencies was not clearly stated.

Chapter 6

6-4, others	The new location of dld.sl has been indicated.
6-17	The system requirements for BIND_DEFFERED and BIND_IMMEDIATE have been clearly stated.
6-19,6-20	The function and use of initializers with dynamic libraries has been added.

Chapter 7

various	Libraries relocated to /usr/shlib.
7-8	The data sizes are indicated by ANSI C types in table 7-4.
7-9	The requirement for the quad routines to be IEEE compliant is stated.

Chapter 8

8-2	Required groups and users listed.
8-8, 8-9	Options for c89 have been modified. The sole purpose of these options is to support the linkage of conforming relocatable objects using conforming shell scripts. All options not directly related to linking have been removed. Of note is the definition of a new option that is not currently supported by HP, the -b option. This option is used to create dynamic libraries. It cannot be used with the -Wl option, as this forces the inclusion of crt0.o.
8-13	Run level 4 (VUE) is deleted. It is not currently a PRO standard.

Chapter 9

9-3	O_NDELAY has been replaced by O_NONBLOCK.
9-4	Initial values for the termios c_cc array are defined.

Chapter 10

10-2	The dynamic library version auxiliary header is only found in relocatable objects, and has been moved to chapter 10 to reflect this fact.
10-10, 10-13	is_first is not used by current systems. It has been replaced by the sort_key field.
10-23	The definition of SS_EXTERNAL was replaced to add clarity.
10-38	R_TRANSLATED is not used on unix systems, and has been deleted.
10-39	Secondary opcodes have been restated as ranges.

Application Binary Interface for PA-RISC Systems

PRECISION RISC ORGANIZATION
Printed in USA October 7, 1993
Draft Edition

Legal Notices

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

PRECISION RISC ORGANIZATION 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.

Precision Risc Organization shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Precision Risc Organization assumes no responsibility for the use or reliability of software or equipment developed to this specification.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

PRECISION RISC ORGANIZATION
19111 Pruneridge Avenue
Cupertino, California 95014 U.S.A.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. The manual printing date indicates its current edition. The printing date changes when a new edition is printed. (Minor corrections which are incorporated at reprint do not cause the date to change.)

1st Draft	January 18, 1993
2nd Draft	June 3, 1993
3rd Draft	August 17, 1993
Final Draft	October 7, 1993

Trademarks

UNIX® is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

X Window System™ is a trademark of the Massachusetts Institute of Technology.

Contents

1

Introduction to the PRO ABI

The PRO *Application Binary Interface* (ABI) defines the structure of binary objects, systems, and the interface between them. Its purpose is to define a binary standard for PA-RISC systems that provides portability of conforming applications across conforming systems without requiring a specific operating system implementation.

A *conforming application* is defined as a binary object or set of objects that use only the system functionality specified by the PRO standards, and access it in the manner stipulated by the standards.

A *conforming system* is defined as an environment that provides, at a minimum, all of the system functionality that is allowed for use in a conforming application.

The PRO Standards

The PRO ABI is designed to be used with the PRO API. The PRO API and PRO ABI are referred to in this document as the PRO Standards.

The PRO API

The PRO *Application Programming Interface* (API) is a meta-standard. It lists industry and de facto standards that specify the behavior of system functionality. The PRO API groups these standards into two major layers as well as into multiple options that may be applied on top of these base layers.

The PRO ABI

This document defines the binary representation of, as well as the interface to the standards listed in the PRO API. It also defines details of the system environment, such as data interchange and file system layout, that are required for binary portability but are not specified in the PRO API standards.

Intended Audience of the PRO ABI

As can be seen in the overview, the majority of the information in this document is of interest to developers of conforming systems and system software (such as compilers).

Elements of the PRO ABI are also relevant to application writers. In particular, chapters 2, 7, 8 and 9 discuss specific restrictions and requirements for application software. The following section contains an overview of these chapters and others.

While these are the primary users currently envisioned for this document, it is anticipated that the PRO ABI and its supplements will be useful to a broader audience.

Overview of the PRO ABI

The PRO ABI defines the system infrastructure that supports the functionality specified in the PRO API, as well as the binary form of the values and data structures required by the core system as defined by the PRO API layers 1 & 2.

The ABI uses the API to identify the functionality to be provided by the system. While the ABI does list all of the supported entry points, it does not define their behavior. The ABI does define implementation specific functionality not included in the API.

Compliance to the PRO API is required of all applications and systems for compliance to the PRO ABI.

The PRO ABI provides two methods for an application developer to produce a conformant binary, *incomplete executables* and *relocatable objects*. Both types achieve portability by placing all system-dependant functionality behind a library interface.

The library interface allows access to the system functionality without imposing undue restrictions on the implementation. An application requests access to library functionality, and the system returns the results, using the conventions specified in this document. The the library entry points must provide all of the functionality specified in the PRO API, but they are allowed to use functionality and conventions not found in the PRO standards.

Incomplete (or dynamically linked) executables are both portable and fully prepared for execution. When executed, the system automatically provides access to the system libraries. If these libraries are updated, no action is required for an application to use the new versions when it is executed. Applications using these *dynamic libraries* are smaller and use less system memory.

Relocatable objects are programs in an unlinked form. They may be *dynamically* linked to provide an incomplete executable, or *statically* linked to provide a complete executable. A complete executable is fully bound and contains no external references. It is larger in size than a dynamically linked program as the library functions have been copied into the program itself. While binding the functionality into the program may increase the applications performance, complete executables are likely to include library code that is not portable and does not conform to the

PRO ABI. Conforming applications are allowed to statically link to any library that conforms to all of the portability requirements of the PRO ABI.

Structure of the PRO ABI

The information in the ABI is grouped into chapters that each deal with a particular aspect of application portability.

Chapter 1 - Introduction to the PRO ABI

This chapter defines the goals, format, and content of the PRO ABI.

Chapter 2 - Software Distribution and Installation

Chapter 2 defines the standards regarding distribution and installation of conforming applications.

Chapter 3 - Low-level System Information

The low-level system information specified in this chapter are the formats and functionality that support or augment the functionality specified by the PRO API.

Chapter 4 - Function Calling Conventions

This chapter specifies the conventions to be used when accessing the system libraries from a conforming application.

Chapter 5 - Object File and Library Formats

Chapter 5 defines the SOM object format as required by executables and dynamic libraries.

Chapter 6 - Program Loading and Dynamic Linking

Chapter 6 describes an incomplete executable and how it is loaded onto a system and attached to the dynamic libraries.

Chapter 7 - Standard Libraries

Chapter 7 lists the standard system libraries, as well as the entry points contained in each.

Chapter 8 - Execution Environment

This chapter defines the standard shell commands, the system environment and the methods a conforming application may use to modify its environment.

Chapter 9 - Terminal and Windowing Interfaces

In general, the details of the terminal and windowing options of the PRO API will be documented in supplements to this ABI. Certain details that must be specified to support the base layers are defined in this chapter.

Chapter 10 - Relocatable Objects

Chapter 10 defines the additional SOM functionality that is required to support relocatable objects and libraries. (Some functionality also may apply to incomplete executables, but it is not required.)

Appendix A - Glossary

Defines terms used in the PRO ABI.

Appendix B - Data Structures and Constants

The standards refer to many data structures, flags, and values. This chapter organizes them using header files and defines their binary representation.

Appendix C - Index

A useful cross-reference.

Conventions

The PRO ABI uses several conventions internally to represent information. These conventions are listed in the sections that follow.

Code Examples

The PRO ABI is intended to be language independent. However, a method is required to represent the information contained. Often, a code example is used to describe a data structure or convention. These code samples are given in either ANSI-C or PA-RISC assembly as defined by the *HP 9000 Computer Systems Assembly Language Reference Manual*. Code examples will be listed using the Courier font.

The code contained in the PRO ABI is listed as a descriptive tool. Conforming systems and applications are not required to use any examples as they appear, but all functional requirements must be met.

Numeric and Character Constants

The ABI specifies constants in many locations. Numeric values that are preceded by the characters "0x" are hexadecimal values. All other numeric values are listed in decimal unless otherwise noted.

ASCII characters and strings are specified within single and double quotes, respectively. Any string termination characters will be explicitly defined and should not be assumed from the representation (unless with a code example).

Registers

Data registers are often referred to using the following abbreviations:

CR	Control Register
FPR	Floating Point Register
GR	General Register
SR	Space Register

Symbols

When referring to a code or data symbol outside of a code example, the symbol will be presented in *italic* form.

Reserved and Undefined Values

When the ABI refers to something as *reserved*, it must not be used by conforming applications or systems. When the ABI refers to a data structure as reserved, the structure must be initialized with null values unless otherwise stated by the ABI.

When the ABI refers to something as *undefined*, it must not be used by conforming applications or systems. When the ABI refers to a data structure as undefined, the structure has no defined semantics and may be initialized with any value.

Related Documents

The PRO ABI was built from, and refers to several standards. These standards include the following:

- The PRO *Application Programming Interface for PA-RISC systems*
- *PA-RISC 1.1 Architecture and Instruction Set Reference Manual, second edition*
- The *ANSI X3.159-1989 Programming Language - C*
- *ISO/IEC 9945-1:1990 Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]*
- *ISO/IEC 646:1991 Information Technology - ISO 7-bit coded character set for information interchange.*
- *ISO 9660: 1988-09-01 Information Processing - Volume and file structure of CD-ROM for information exchange.*
- *European Computer Manufacturers Association (ECMA) 139 - June 1990*
- Unix System Laboratories, *System V Application Binary Interface*
- *System Interface Definitions, Issue 4* of the X/Open CAE Specification
- *HP 9000 Computer Systems Assembly Language Reference Manual, 4th edition*
- The *X Window System*™ graphical user interface specification
- The *Precision Risc Organization By-Laws*

Modifications to the PRO ABI

The PRO standards are developed under the authority of the PRO Technology Council and approved by the PRO Board of Directors. Proposals to modify these standards should be presented to the Technology Council directly or through one of the Special Interest Groups. Consult the *Precision Risc Organization By-Laws* for more information.

Future versions of this document will list all changes from previous editions.

2

Software Distribution & Installation

Without the ability to load an application onto a system, there is no reason to ensure the two are compatible. The ABI does not specify specific user instructions to be used for the installation of conforming applications. Each application is responsible for documenting the procedures for its own installation. Any automation provided to aid installation (programs, shell scripts) must conform to the PRO Standards.

This chapter defines the distribution media to be used by a conforming application, as well as the file system structure provided to support applications.

Software Installation and Packaging

Currently, a wide variety of media types are in general usage. This is an inhibitor to software portability due to the cost of supporting all types for applications or systems. The PRO ABI does not intend to restrict the use of these formats, but has selected two to be required formats.

Installation Utilities

Only 3 utilities for reading installation archives are supported for use on conforming systems, *tar*, *cpio*, and *pax*. A conforming application is allowed to provide additional utilities to be used for installation with the restrictions:

- All utilities must be certified as conforming to the PRO standards.
- The utilities must be loadable using tar, cpio, or pax.
- Any temporary storage conforms to the requirements of the next section.

Note

Of the listed utilities, pax is recommended for use. Both tar and cpio are currently marked for withdrawal by the PRO API.

Media Formats

Conforming applications must be available on CD-ROM or DDS tape as defined below. A conforming system must be able to support devices that can work with either formats. A conforming application must use at least one of these formats.

CD-ROM

CD-ROM must conform to *ISO 9660:1988*, as specified in *Information Processing-Volume and File Structure of CD-ROM for Information Interchange*, and a conforming system must support the level 1 implementation and the level 2 interchange of the *ISO 9660* standard. A CD-ROM may be accessed as a raw device, or mounted as a file system to be accessed directly.

DDS Tape

Digital Audio Tape (DAT) must conform to the European Computer Manufacturers Association (ECMA) Digital Data Storage (DDS) ECMA 139 June 1990 standard.

Data Formats

Conforming systems must support both the Extended tar and Extended cpio formats, as defined by *ISO/IEC 9945-1:1990* (also known as *ANSI/IEEE POSIX Standard 1003.1-1990*).

Directory Structure for Application Use

Certain sections of the directory structure have been allocated for use by conforming applications. The structure selected is based on the layout specified in the *System V Application Binary Interface*. The file system is organized in a manner that separates applications from the operating system.

This section is primarily concerned with the file system as used by an application during installation. Additional information regarding the file system found on a conforming system is provided in "Required Files and Directories" on page 8-3

In the directories described below, the term *<application>* is used as a marker. Each application should replace this marker with a unique name that is associated with the application and a legal POSIX directory name.

/etc/opt/*<application>*

This directory is used to hold machine-specific configuration files used by the conforming application.

/etc/opt/profile.d

/etc/opt/csh.login.d

These directories hold symbolic links to shell initialization scripts located in the applications main directory. For a full description, see "User-shell Initialization" on page 8-13.

/opt/*<application>*

This directory is intended to hold all of the static files used by the application. The value of *<application>* should reflect the application in question.

/opt/<*application*>/bin

This directory is intended to hold all executables provided with the application. If present, it should be added to the PATH list in the shell initialization scripts. The use of this directory is recommended, but not required.

/opt/<*application*>/lib

This directory is intended to hold any libraries provided with the application. The use of this directory is recommended, but not required.

/opt/<*application*>/newconfig

This directory is intended to hold the default configuration data files provided with the application. The use of this directory is recommended, but not required.

/opt/<*application*>/lib/nls

This directory is intended to hold any NLS catalogs provided with the application. The use of this directory is recommended, but not required.

/opt/<*application*>/man

This directory is intended to hold any man pages provided with the application. If present, it should be added to the MAN_PATH list in the shell initialization scripts. The use of this directory is recommended, but not required.

/var/opt/<*application*>

This directory is reserved for applications to use with variable data, such as temporary and log files.

/var/preserve

/var/preserve is intended to hold temporary files generated by editors.

/var/tmp

/var/tmp holds temporary files, including those generated by applications.

/sbin/init.d

/sbin/rc#.d

These two directories are used with system initialization scripts. The scripts reside in the /sbin/init.d directory, but execution is through symbolic links located in the /sbin/rc#.d directories. For more information, see "System Initialization Shell Scripts" on page 8-10.

/tmp

The /tmp directory is for the use of system generated temporary files and its contents are not usually preserved across a system reboot. Applications should use /var/tmp or /var/opt/*<application>* for storage of working files.

3

Low-level System Information

In order for systems and applications to be developed in a compatible manner, basic conventions and definitions must be established which detail the interactions and data that are shared between applications and systems.

This chapter specifies information that is used below the level at which most application developers are expected to work, but is required for portability. This information includes the subset of the PA-RISC 1.1 architecture that a conforming application is permitted to use, how data is represented at the binary level, and how an application interacts with a conforming system.

Machine Interface

The PA-RISC architecture is defined in the *PA-RISC 1.1 Architecture and Instruction Set Manual*, and the PRO ABI does not attempt to reproduce its full content. It does define how data is represented on PA-RISC systems as well as describing the subset of the architecture that is supported for use by conforming applications.

Instruction Set

This section specifies the processor and floating-point coprocessor instructions that are available to a conforming application.

While executing these instructions, a conforming application must satisfy and can rely on the requirements specified in chapter 4 of the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*:

- "Atomicity of Storage Accesses"
- "Ordering of Accesses"
- "Completion of Accesses"
- "Instruction Pipelining"
- "Branching"

Processor Architecture

The processor instruction set is defined in Chapter 5 of the *PA-RISC 1.1 Architecture and Instruction Set Manual*. Processor instructions that are available to a conforming application are listed in Table 3-1. The system developer may determine whether the instructions are implemented in software or hardware.

Note

The effect, on a conforming application, of executing any other processor instruction (other than the floating-point coprocessor instructions described in the next section) is undefined.

Table 3-1: Processor Instructions Available to Conforming Applications

ADD	ADDBF	ADDBT	ADDC	ADDCO
ADDI	ADDIBF	ADDIBT	ADDIL	ADDIO
ADDIT	ADDITO	ADDL	ADDO	AND
ANDCM	BB	BE	BL	BLE
BLR	BREAK	BV	BVB	COMBF
COMBT	COMCLR	COMIBF	COMIBT	COMICLR
DCOR	DEP	DEPI	DS	EXTRS
EXTRU	FDC	FIC	IDCOR	LDB
LDBS	LDBX	LDCWS	LDCWX	LDH
LDHS	LDHX	LDIL	LDO	LDSID[1]
LDW	LDWM	LDWS	LDWX	MFCTL[1]
MFSP[1]	MOVB	MOVIB	MTCTL[1]	MTSP[1]
OR	PDC	PROBER	PROBERI	PROBEW
PROBEWI	SH1ADD	SH1ADDL	SH1ADDO	SH2ADD
SH2ADDL	SH2ADDO	SH3ADD	SH3ADDL	SH3ADDO
SHD	STB	STBS	STBYS	STH
STHS	STW	STWM	STWS	SUB
SUBB	SUBBO	SUBI	SUBIO	SUBO
SUBT	SUBTO	SYNC	UADDCM	UADDCMT
UXOR	VDEP	VDEPI	VEXTRS	VEXTRU
VSHD	XOR	ZDEP	ZDEPI	ZVDEP
ZVDEPI				

1. These instructions access control registers or space registers. Follow the appropriate register restrictions described in the section "Processor Resources" later in this chapter.

Floating-Point Coprocessor Instructions

Floating-point coprocessor instructions are implementations of the basic coprocessor instructions with the *uid* value restricted to 0 or 1. The floating-point coprocessor instruction set is defined in Chapter 6 of the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.*

A conforming application may use any of the floating-point coprocessor instructions shown in Table 3-2.

Note

Support for quad precision operations is provided by the library routines defined in "Quad-precision Emulation Routines" on page 7-9.

Table 3-2: Floating-Point Coprocessor Instructions Available to Conforming Applications

COPR,0,0	FABS[1]	FADD[1]	FCMP[1]	FCNVFF[1]
FCNVFX[1]	FCNVFXT[1]	FCNVXF[2]	FCPY[1]	FDIV[1]
FLDDS	FLDDX	FLDWS	FLDWX	FMPY[1]
FMPYADD[1]	FMPYSUB[1]	FRND[1]	FSQRT[1]	FSTDS
FSTDX	FSTWS	FSTWX	FSUB[1]	FTEST[1]
XMPYU[2]				

1. These instructions must follow the restrictions for accessing the floating-point registers as described in the "Processor Resources" section below. In a conforming application, these instructions must operate only on single and double precision floating-point data.
2. These instructions must follow the restrictions for accessing the floating-point registers as described in the "Processor Resources" section below. In a conforming application, these instructions must operate on integer data only.

Processor Resources

The general, control, and space registers are described in Chapter 2 and the floating-point coprocessor registers are described in Chapter 6 of the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* The following restrictions apply to their use by a conforming application.

General Registers

A conforming application can load into, store from, or operate on any of the general registers (GR 0 through GR 31) subject to the conventions specified in chapter 4.

Space Registers

A conforming application can move to and from space registers 0 through 3 (SR 0-3) using the MOVE TO SPACE REGISTER (MTSP), MOVE FROM SPACE REGISTER (MFSP) and LOAD SPACE IDENTIFIER (LDSID) instructions.

A conforming application can move *from* space registers 4 through 7, but must not move *to* space registers 4 through 7. Space register 4 is unprivileged, but must not be changed from the value established by the system. Space registers 5 through 7 are privileged and cannot be changed by a conforming application.

Control Registers

A conforming application may move to and move from control register 11 (the Shift Amount Register) using the MOVE TO CONTROL REGISTER (MTCTL) and MOVE FROM CONTROL REGISTER (MFCTL) instructions.

Even though control register 16 (the Interval Timer) is unprivileged and can be read when the PSW S-bit is zero, a conforming application must never move from control register 16. Control registers 26 and 27 are currently reserved for future use.

Conforming applications may not access any other control registers, nor move to control register 16. Such accesses can only be performed by privileged software.

Floating-Point Coprocessor Registers

A conforming application can load or store single-words into or from floating-point registers 4L through 31L and 4R through 31R. A conforming application can also load or store doublewords into floating-point registers 4 through 31.

A conforming application can load or store a single word or doubleword into or from floating point register 0 (the Floating Point Status Register). Bits 32 through 63 of the result of a double-word store are not specified. When used as a source operand of a floating-point operation, FPR 0 provides the value 0.0 of type double.

A conforming application must not load or store single-words or doublewords into any other floating-point register.

A conforming application may operate on floating-point registers, 0L, 4L through 31L, 4R through 31R, 0, and 4 through 31 using the floating-point coprocessor operation instructions.

Data Representation

The PRO ABI does not attempt to define the rules for all programming languages. The PRO ABI uses ANSI C to describe the data structures used when interacting with conforming systems. The following sections are meant to define ANSI C sufficiently to support its use as a descriptive tool.

Character Representations

Exchanges between conforming applications and the system shall interpret character values in the range 0 through 127 in accordance with the 7-bit ISO/IEC 646:1991 standard International Reference Version encoding. These exchanges include all external references and file names.

The Precision Risc Organization will publish localization standards in the form of separate documents as they are developed.

Byte Ordering

Conforming applications must use Big Endian byte ordering. Bytes are numbered from left to right in bytes, halfwords, words, or doublewords. Bits are also numbered from left to right in each of these elements. The most significant byte or bit in one of these elements has the lowest address, and the least significant bit or byte has the highest address. The following figure shows the numbering of bits and bytes in a word of data.

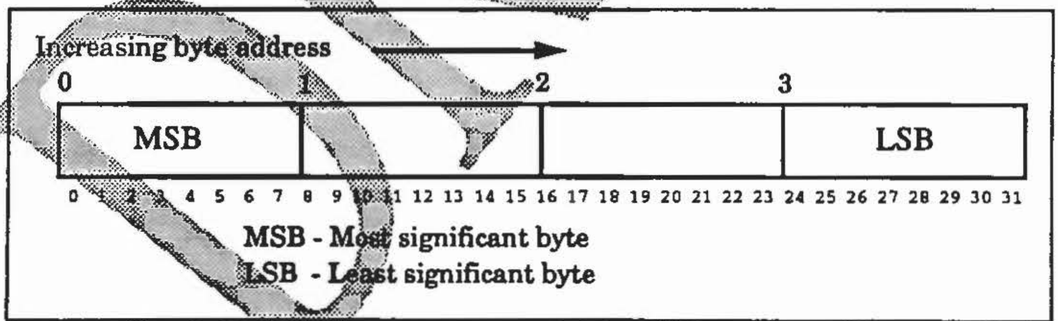

Figure 3-1: Ordering of Bytes and Bits within a Word of Data

Primitive Data Types

Table 3-3 shows the correspondence between processor primitive data types and ANSI C scalar types. Data types for other languages will be defined relative to the ANSI C types. When a data object of a given primitive type is stored in memory, it is expected that it will be aligned as specified in the table. If it is not aligned and the conforming system does not handle the trap in a manner transparent to the application, then the conforming application receives a SIGBUS signal.

Table 3-3: Primitive Data Types Available to a Conforming Application

Primitive Type	Size in Bytes	Alignment in Bytes	ANSI C Type
byte	1	1	`signed char` `char`
unsigned byte	1	1	`unsigned char`
halfword	2	2	`short` `signed short`
unsigned halfword	2	2	`unsigned short`

word	4	4	`int`
			`signed int`
			`long`
			`signed long`
			`signed long int`
			`enum`
			`anytype*`
			`anytype(*)()`

unsigned word	4	4	`unsigned int`
			`unsigned long`
			`unsigned long int`

single binary floating-point	4	4	`float`
double binary floating-point	8	8	`double`
quad precision floating-point	16	16	`long double`

Alignment of Data Elements

This section specifies the alignment of the data elements of a conforming application. If a pointer is described as pointing to a particular data type, the implications are that the value of the pointer is an address that is a multiple of the size of that data type, and that storage of sufficient size is available at that address.

Aggregates and Unions

The allocation of aggregates (structures and arrays) and unions and the alignment of their components are subject to the following rules.

- Aggregates and unions are aligned on a boundary that is a multiple of the alignment for their most strictly aligned component.
- The components of a structure are arranged in their order of appearance in the structure declaration.
- Each component of a structure is aligned on a boundary that is a multiple of the component's alignment. Internal padding bytes are added after a component, if necessary, to preserve the alignment of the following component.
- The size of a structure, union, or array is a multiple of the strictest alignment boundary of its components.
- Padding bytes are added, if necessary, to the end of a structure to make it the proper size.
- The contents of any padding bytes are unspecified.

Bit Fields

ANSI C "`struct`" and "`union`" definitions may have bit fields defining integral objects with a specified number of bits. Bit fields always have non-negative values unless "signed" is specified. Bit fields obey the same size and alignment rules as other aggregate and union components with the following additions.

- Bit fields are allocated from left to right (most to least significant bit).
- A bit field may begin on a bit boundary within a storage unit, but it must reside entirely within its base type. Thus, a bit field never crosses the unit boundary of its allocation type. Padding may be added before the

bit field to preserve this alignment restriction. ANSI C allows the base type to be int, unsigned int, or signed int.

- Bit fields may share a storage unit with as many other "`struct`" components as there is room to accommodate.
- The size of a structure containing bit fields is a multiple of the alignment size of the type with the strictest requirements. Padding bits and bytes are added at the end of a structure to make it the proper size.
- A bit field of zero length specifies that no additional bit fields may share space in any preceding storage unit.
- The type of a bit field has no further effects on the alignment of a structure or union.

Figure 3-3 shows the layout of a data structure of type `struct a` as defined in Figure 3-2.

```
struct t {                          struct a {
        char    u[3];                       int     b:4;
        int     v:10;                       char    c;
        int     w:10;                       char    d;
        int     x:12;                       int     e:22;
        int     y:2;                        int     :0;
        double  z;                          short   f;
}                                           struct t  g;
                                    }
```

Figure 3-2: Example data structure

<table>
<tr><th>word addr</th><th colspan="2">byte 0</th><th>byte 1</th><th>byte 2</th><th>byte 3</th></tr>
<tr><td>0</td><td>b</td><td>pad[1]</td><td>c</td><td>d</td><td>pad[2]</td></tr>
<tr><td>1</td><td colspan="4">e</td><td>pad[3]</td></tr>
<tr><td>2</td><td colspan="3">f</td><td colspan="2">pad[4]</td></tr>
<tr><td>3</td><td colspan="5">pad[4]</td></tr>
<tr><td>4</td><td colspan="2">u[0]</td><td>u[1]</td><td>u[2]</td><td>pad[5]</td></tr>
<tr><td>5</td><td colspan="2">v</td><td colspan="2">w</td><td>x</td></tr>
<tr><td>6</td><td>y</td><td colspan="4">pad[6]</td></tr>
<tr><td>7</td><td colspan="5">pad[6]</td></tr>
<tr><td>8
9</td><td colspan="5">z</td></tr>
</table>

Figure 3-3: Layout of example data structure

1. The bitfield *b* does not use all of the allocated storage area. Padding is added to align the next element, char *c*, on a byte boundary.
2. The bitfield *e* does not fit in the unused portion of the preceding storage unit and a new storage unit must be allocated and aligned.
3. The unnamed bitfield of zero width closes the preceding storage unit to further use.
4. The structure *g* must be aligned according to the strictest alignment of any of the structure members, in this case, *z*.
5. The bitfield v cannot fit in the remaining space and a new storage unit must be allocated and aligned.
6. Padding must be added for the double *z* to be aligned on a double-word boundary.

Operating System Interface

Virtual Address Space

The processor's 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.

Each conforming application has its own short address space composed of four quadrants; the format is shown in Figure 3-4. For more detail on short pointers and addressing, see Chapter 3 of the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.*

The first quadrant of a SHARE_MAGIC files short address space is mapped by space register 4 to the first quadrant of a space containing the shared text of the conforming application. The text is readable and executable, but not writable and must begin at a page boundary (a page is defined to be 4096 bytes in size). A conforming application must not change the contents of SR 4.

The second quadrant of the short address space is mapped by space register 5 to the second quadrant of a space containing the private data of a conforming application. All allocated space in the second quadrant 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. Conforming systems must support a minimum of 8Mbytes for the user stack.

In a SHARE_MAGIC executable, the first page of the second quadrant is reserved as the guard page. This requires that the data segment may not be placed lower than 0x40001000.

The third and fourth quadrants of the short address space are mapped by space registers 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 have the permissions granted by the creating calls.

The first page of the fourth quadrant is called the Gateway Page. It contains the code used to perform the supported system calls. It must be both readable and executable by conforming applications.

A DEMAND_MAGIC file is identical in layout to files of the SHARE_MAGIC type. They differ in how they are loaded at run-time. Code in a SHARE_MAGIC file is loaded completely when executed, while code in a DEMAND_MAGIC file is loaded as needed during execution.

Conforming systems may use space in any quadrant that has not been previously reserved for use by a conforming system.

An EXEC_MAGIC executable differs from a SHARE_MAGIC file in that the first two quadrants (quadrants 0 & 1) must reside in the same space. This allows data to cross the quadrant boundary and begin at the first full page following the program text (code).

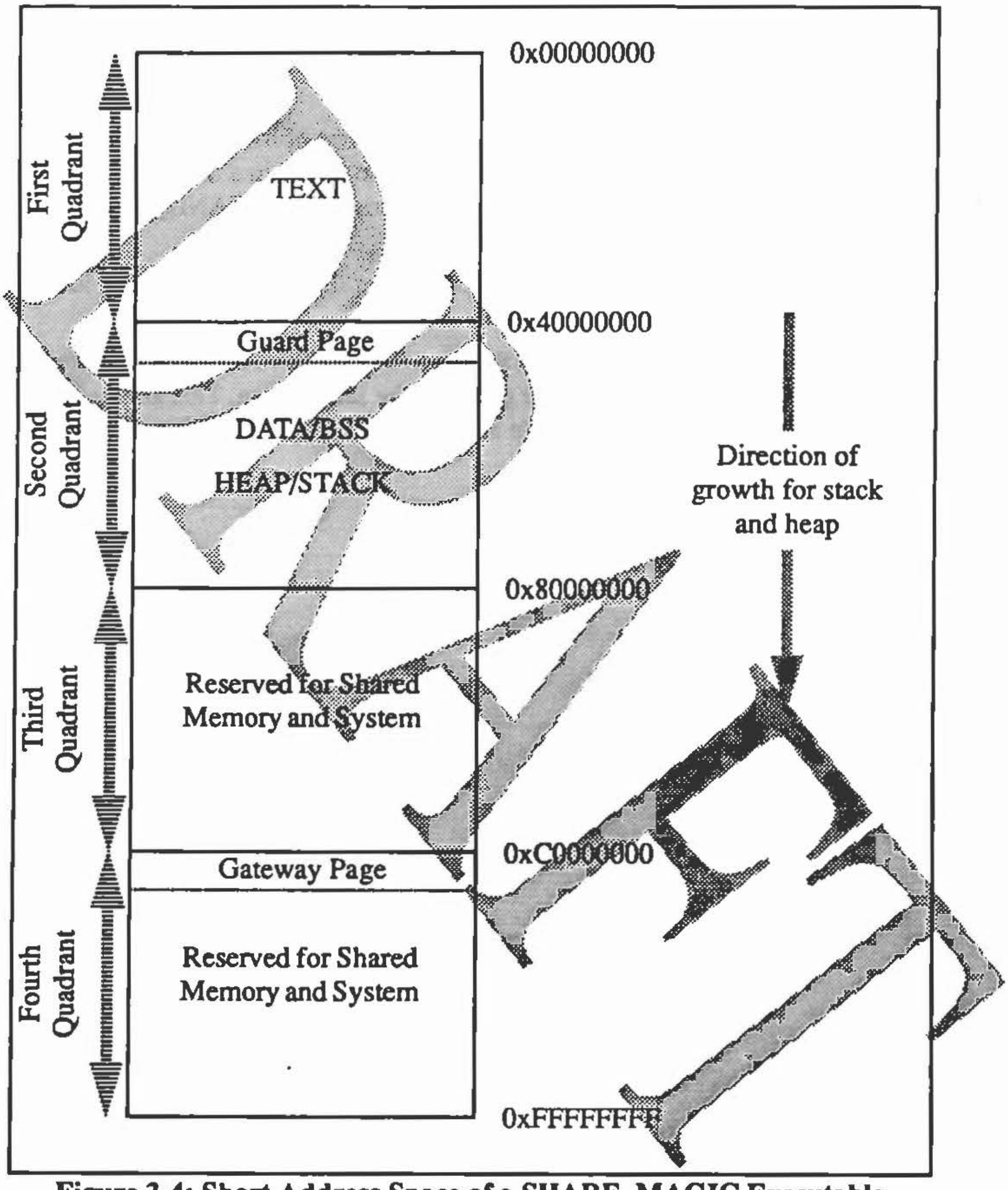

Figure 3-4: Short Address Space of a SHARE_MAGIC Executable

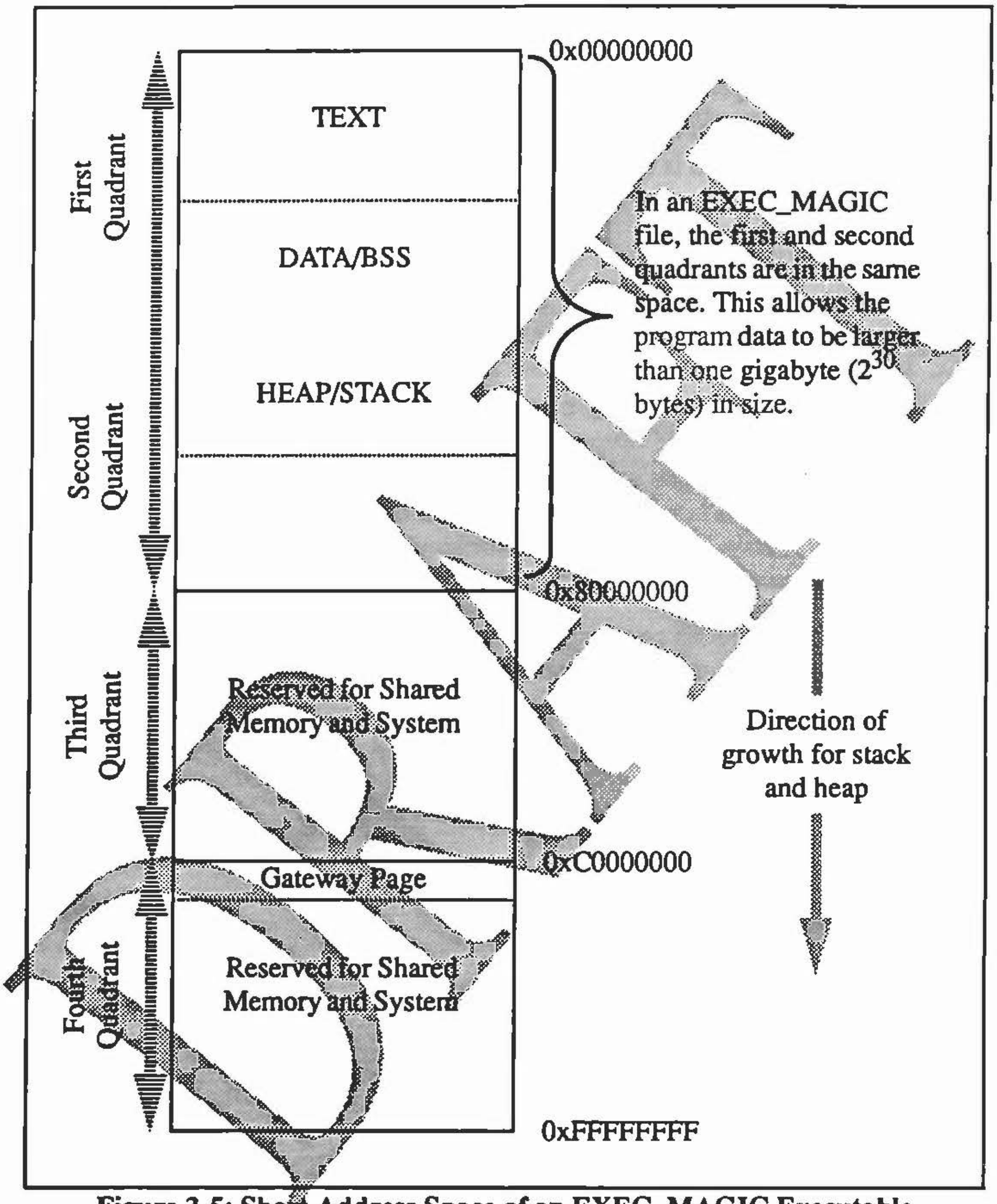

Figure 3-5: Short Address Space of an EXEC_MAGIC Executable

Processor Execution Modes

The *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* states that there are 4 privilege levels (0-3) at which code may be executed. Applications may only run at privilege level 3 (the lowest level).

A conforming application is not allowed use of the GATEWAY instruction.

Exception Interface

The system may detect faults that affect the execution of an application. These interruptions and the events that cause them are discussed in Chapter 4 of the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.*

If an interruption occurs during the execution of a conforming application, the system software can respond in one of three ways:

- A signal is sent to the application
- The interruption is handled in a way that is transparent to the application
- The application is affected in an undefined way, often resulting in the termination of the conforming application.

The effects of interruptions on a conforming application are shown in Table 3-4. Interruptions other than those listed have no effect on a conforming application.

Table 3-4: Effects of Interruptions of Conforming Applications

Interruption	Signal
High-priority machine check	Effect not specified
Power failure interrupt	SIGPWR or effect not specified[1]
Instruction TLB miss fault/Instruction page fault	SIGSEGV or none[2]
Instruction memory protection trap	SIGSEGV, SIGBUS or none[3]
Overflow trap	SIGFPE[4]
Conditional trap	SIGFPE[4]
Assist exception trap	SIGFPE[4] or none[5]
Data TLB miss fault/Data page fault	SIGSEGV or none[2]
Non-access instruction TLB miss fault	SIGSEGV or none[6]
Non-access data TLB miss fault/Non-access data page fault	SIGSEGV or none[6]
Data memory access rights trap[7]	SIGSEGV, SIGBUS or none[8]
Data memory protection ID trap[7]	SIGSEGV, SIGBUS or none[8]
Unaligned data reference trap[7]	SIGSEGV, SIGBUS or none[8]
Data memory break trap	Effect not specified
Assist emulation trap	SIGFPE[4]

1. If a system supports powerfail recovery, a power failure interrupt causes a conforming application to receive a SIGPWR signal after power is restored. If the system does not support powerfail recovery, the effect of a power failure interrupt on a conforming application is not specified.

2. If the fault is for a valid page belonging to the conforming application and access rights for the page allow the attempted operation, the conforming application does not receive a signal. Otherwise, the conforming application receives a SIGSEGV signal. Unallocated pages in an applications stack may be referenced by a conforming application. An exception exists for memory-mapped files. Attempts to access any full page between the end of the file and the end of the mapped file region shall result in the application receiving a SIGBUS signal.
3. If the trap occurs on a valid page owned by the conforming application and access rights for the page allow the attempted operation, the conforming application does not receive a signal. If the page is not owned by the conforming application, then it receives a SIGSEGV signal. If the access rights for the page do not allow the attempted operation, the conforming application receives a SIGBUS signal.
4. To conform to IEEE-785 standards, it must be possible to identify the specific type of fault that caused the SIGFPE to be raised. This ability is provided through the floating-point status register (FPR 0) value passed in the *save_state* structure (under *sigcontext/siglocal*).
5. Floating-point instructions are defined in the PA-RISC instruction set, but may be implemented in the processor hardware or emulated by system software. If an assist exception trap is received because of an unimplemented, valid floating-point instruction, then the conforming application does not receive a signal. Instead, system software emulates the instruction before returning control to the conforming application. All other causes of the assist exception trap result in the conforming application receiving a SIGFPE signal.
6. Traps caused by the FDC & FIC instructions behave as reads and writes. Traps caused by the PROBE instructions must be handled by the conforming system with no signal sent to the conforming application.
7. The data memory access rights trap, data memory protection trap and unaligned data reference trap were combined into a single data memory protection trap/unaligned data reference trap in earlier versions of the PA-RISC architecture.
8. If the trap occurs on a valid data page owned by the conforming application, and the access rights of the page and the alignment of the data allow the attempted operation, then the conforming application does not receive a signal. If the page is not owned by the conforming application, then it receives a SIGSEGV signal. If the access rights for the page do not allow the attempted operation, or an attempt is made to reference unaligned data, then the conforming application receives a SIGBUS signal.

Signaling a Conforming Application

Signals may be generated by the system, or applications may use functions such as kill() and raise() to programmatically send a signal. For these signals to be recognized by conforming applications, a standard convention for signal passing must be defined.

A signal handler is called with three parameters; the signal number, a code field (usually provided by hardware), and a pointer to the *sigcontext* structure. These parameters will be passed as shown in Table 3-5.

The signals that are currently defined for use on conforming PA-RISC systems are specified in "<signal.h>" on page B-25.

The code field is always zero, except for the SIGILL and SIGFPE signals. These code values are listed in Table 3-6 and shall be set by a conforming system when processing an interruption. In all other cases, the code field shall be set to 0.

The *sigcontext* structure must be defined by the system when a signal is sent. The pointer to this structure is valid only during the context of the signal handler.

Table 3-5: Register Usage When Signaling a Program

Register	C Source Definition	Usage
GR 2		return pointer
GR 24	`struct sigcontext * scp`[1]	pointer to context structure
GR 25	`int code`	information code
GR 26	`int sig`	signal number
GR 30		stack pointer

1. The sigcontext structure and the code field are not defined by the PRO API signal routines. These fields are defined only for signals generated by a conforming system.

Table 3-6: Defined Values for Signal Codes

Signal	Code Value	Description
SIGILL	8	illegal instruction trap
	9	break instruction trap
	10	privileged operation trap
	11	privileged register trap
SIGFPE	12	overflow trap
	13	conditional trap
	14	assist exception trap
	22	assist emulation trap

System Calls

The system call interface is implementation-specific and is intended to be obscured by the library interface, however, a small set is required to support dynamic linking. Conforming applications may not use direct system calls other than as required to support program loading.

Table 3-7 lists all supported system calls and the details for each call.

Invoking a System Call

The following three steps must be followed to invoke one of the supported system calls:

1. Place any arguments in the proper locations.
2. Place the system call number in GR 22.
3. Branch to the gateway location (address 0xC0000004 in the space identified by SR 7), leaving the return pointer in GR 31.

Values Returned by a System Call

All system calls return a status indication in general register GR 22. When the value returned in GR 22 is the integer value 0, the system call completed successfully. When the value returned in GR 22 is a value other than 0, the system call failed.

If the system call completed successfully, then the returned value is located in general register GR 28. The tables that appear below specify particular data type. If the system call failed, then the external variable `errno` may provide additional information about the cause of the failure. In all the cases when the operating system documentation specifies that error information is provided, the value for `errno` is located in GR 28., Values for `errno` are specified in <errno.h>.

Table 3-7: System Calls

Call Name	Call No.	Arguments			Returned Value
		General Register	Type	Variable Name	Type
`close`	6	GR 26	`int`	`fildes`	`int`
`_exit`	1	GR 26	`int`	`status`	`void`

`lseek`	19	GR 26
		GR 25
		GR 24
			`int`
			`off_t`
			`int`
				`filedes`
				`offset`
				`whence`
					`off_t`

`mmap`	71	GR 26
		GR 25
		GR 24
		GR 23
		[SP - 52][1]
		[SP - 56][1]
			`caddr_t`
			`size_t`
			`int`
			`int`
			`int`
			`off_t`
				`addr`
				`len`
				`prot`
				`flags`
				`fildes`
				`off`
					`caddr_t`

`open`	5	GR 26
		GR 25
		GR 24
			`char *`
			`int`
			`mode_t`
				`path`
				`oflag`
				`mode`
					`int`

`read`	3	GR 26
		GR 25
		GR 24
			`int`
			`void *`
			`size_t`
				`fildes`
				`buf`
				`nbyte`
					`ssize_t`

`write`	4	GR 26
		GR 25
		GR 24
			`int`
			`void`
			`size_t`
				`fildes`
				`buf`
				`nbyte`
					`ssize_t`

1. Only four registers may be used for argument passing. Additional arguments must be passed on the stack. Refer to Table 4-1, on page 4-5 for more information.

4

Function Calling Conventions

It is the goal of the PRO ABI to enable applications to be portable across different implementations of PA-RISC systems without excessive restrictions on the implementation of those systems. To achieve this, access to system functionality is only allowed through libraries local to each implementation.

System libraries are not intended to be portable, and code local to an application can be structured in any way that does not violate the restrictions of the PRO ABI. The interface between applications and the system libraries must be consistent across systems and applications. This section defines the requirements of this interface.

Stack Usage

Because no explicit procedure call stack exists in the PA-RISC processor architecture, the stack is defined and manipulated entirely by software convention. When a process is initiated by the operating system, a virtual address range is allocated to that process to be used for the call stack. Conforming systems must allow a minimum stack size of 8 Mbytes. 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 procedures frame to exist at the addresses below the stack pointer. As procedures are exited, the stack pointer is decremented by the same amount. The stack pointer always points to the first word above the current frame.

Leaf and Non-Leaf Procedures

All procedures can be classified in one of two categories: leaf or non-leaf. A leaf procedure is one that makes no additional calls, while a non-leaf procedure is one that does make additional calls. Although simple, the distinction is essential because the two cases entail considerably different requirements regarding (among other things) stack allocation and usage. Every non-leaf procedure requires the allocation of an additional stack frame in order to preserve 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.

Storage Areas Required for a Call

It is often the case that much of a procedure's state information is saved in the caller's frame. This helps to avoid unnecessary stack usage. A general picture of the top of the stack for one call, including the frames belonging to the caller (previous) and callee (new) is shown in Figure 4-1.

The elements of a single stack frame that must be present in order for a procedure call to occur are shown in Table 4-1. 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.

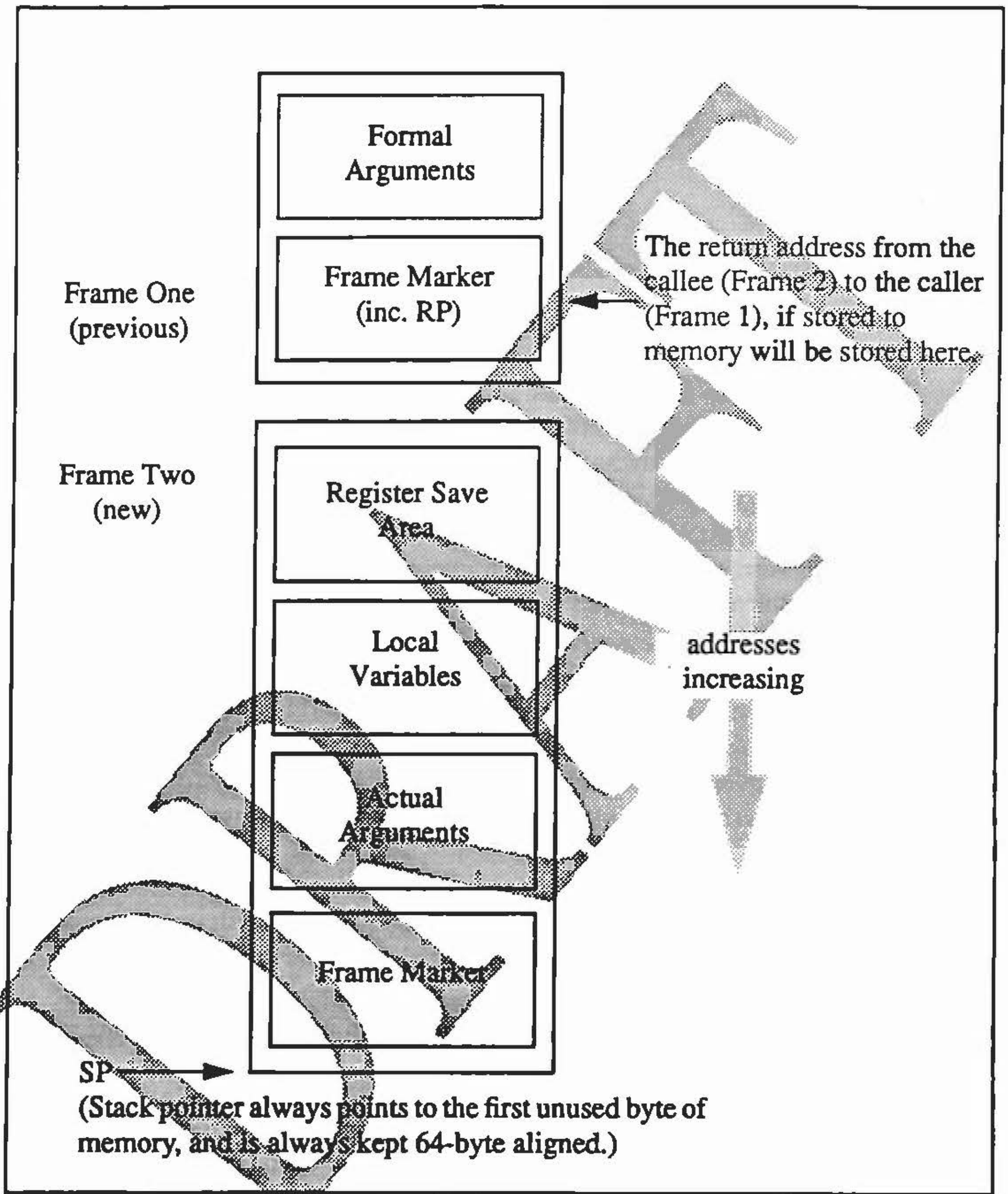

Figure 4-1: General Stack Layout

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. See "Parameter Passing and Function Results" on page 4-14. for detailed discussion of Parameter Relocation stubs.

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

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

Current RP: Reserved for use by the called procedure; this is where the current return address must be stored to support stack unwinding if the procedure uses RP (GR2) for any other purpose.

External/Stub RP (RP'), External SR4/LTP', and External DP/LTP: All three of these words are reserved for use by the inter-modular (external) calling mechanism. See "External Calls" on page 4-19. for more details.

Table 4-1: Elements of Single Stack Frame Necessary for a Procedure Call

Offset	Contents	
Variable Arguments (optional; any number may be allocated)		
SP-(4*(N+9))	arg word N	
:	:	
:	:	
SP-56	arg word 5	
SP-52	arg word 4	
Fixed Arguments (must be allocated; may 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 >	

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 the diagram, there may actually be an unlimited number of arguments stored on the stack, continuing downward in succession (with addresses that correspond to the expression given in the diagram). Any necessary allocation in this area must be made by the caller.

Register Usage and Parameter Passing

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

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.

The register layouts are shown in Figure 4-2 and Figure 4-3.

Note

In Figure 4-3, the conventions listed apply to both the single- and double-word floating point registers.

Register	Usage
GR 0	Value (zero)
GR 1	Scratch *
GR 2	RP (Return Pointer/Address)
GR 3 : : GR 18	Callee Saves
GR 19 : GR 22	Caller Saves
GR 23 : GR 26	Arguments *
GR 27	DP (Global Data Pointer)
GR 28, GR 29	Return Values *
GR 30	SP (Stack Pointer)
GR 31	MRP (Millicode Ret. Ptr)/Scratch*

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

Figure 4-2: General Register Partitioning

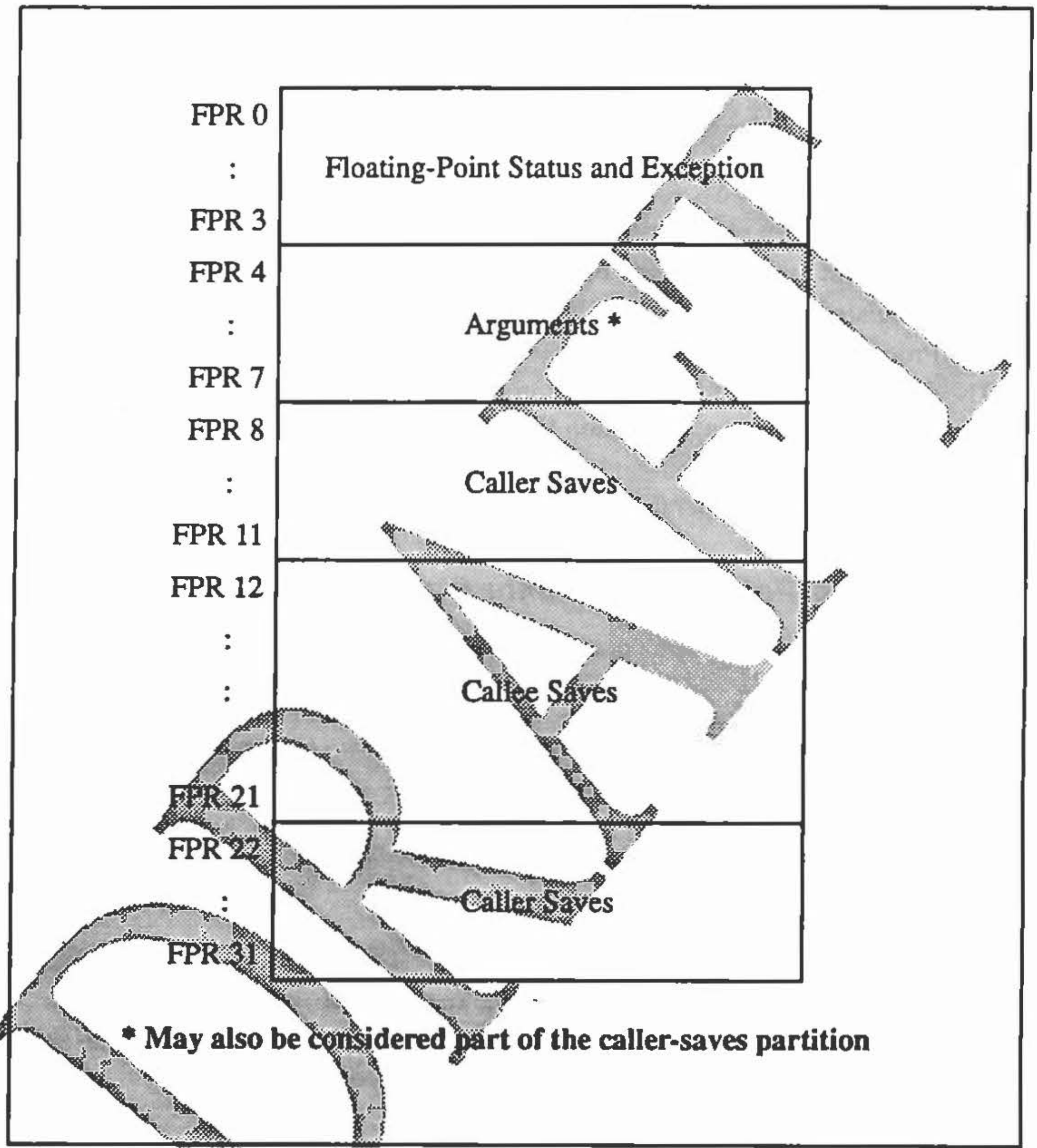

Figure 4-3: Floating-Point Register Partitioning

Other Register Conventions

The following are guaranteed to be preserved across calls:

- The procedure entry value of SP.
- The value of DP.
- Space registers SR 3, SR 4, SR 5, SR 6, and SR 7.

The following are not necessarily preserved across calls:

- Floating-point registers FPR 1, FPR 2, and FPR 3.
- Space registers SR 0, SR 1, and SR 2.
- The Processor Status Word (PSW).
- The shift amount register (CR 11) or any control registers that are modified by privileged software (e.g. Protection IDs).
- The state, including internal registers, of any special function units accessed by the architected SPOP operations.

The Floating-Point Coprocessor Status Register

Within the floating-point coprocessor status register (FPR 0), 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.

Summary of Dedicated Register Usage

Table 4-2, Table 4-3, and Table 4-4 show the required conventions regarding argument and return value passing. Conforming applications must conform to all procedure calling conventions specified in the PRO ABI.

Note

If the routine in question is a non-leaf routine, GR 2 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 GR 2 be restored before exit; a branch (BV) using another caller-saves register is allowed.

Table 4-2: Space Register Usage

Register Name	Other Names	Usage Convention
SR 0		Caller-saves space register or millicode (nested or external) 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.

Table 4-3: 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).
GR 2	RP	Return pointer and scratch register.
GR 3 - GR 18		General purpose callee-saves registers.
GR 19		Dynamic Library linkage register.
GR 19 - GR 22		General purpose caller-saves registers.
GR 22		System calls. (See "System Calls" on page 3-21.
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		Scratch register (caller-saves).

Table 4-4: Floating-Point Usage

Register Name	Other Names	Usage Convention
FPR 0		Floating-point coprocessor status. See discussion in the text of this chapter.
FPR 1 - FPR 3	∅	Floating-point exception registers. Cannot be modified by user code.
FPR 4	fret farg0	Floating-point return register, single-precision argument register 0, or floating-point caller-saves register.
FPR 5	farg1	Single-precision argument register 1, double-precision argument register 0 or floating-point caller-saves register.
FPR 6	farg2	Single-precision argument register 2,or floating-point caller-saves register.
FPR 7	farg3	Single-precision argument register 3, double-precision argument register 1 or floating-point caller-saves register.
FPR 8 - FPR 11		Floating-point caller-saves registers.
FPR 12 - FPR 15		Floating-point callee-saves registers.
FPR 16 - FPR 21		Floating-point callee-saves registers, only available on PA-RISC version 1.1 or later processors.
FPR 22 - FPR 31		Floating-point caller-saves registers, only available on PA-RISC version 1.1 or later processors.

Parameter Passing and Function Results

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 must be extended to 32 or 64 bits, by right-justifying the value itself, and then left-extending it. Non-standard length parameters that are signed integers are sign-extended to the left to 32 or 64 bits.

Table 4-5 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 4-5: 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	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 SR 4 - SR 7) 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.

Reference Parameters

A short pointer to the referenced data item (using SR 4-SR 7) 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.

Routine References

This convention requires that routine references (i.e. procedure parameters, function pointers, external subroutines) be passed as 32-bit nonfloating-point plabels. See "Procedure Labels and Dynamic Calls" on page 4-26.

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 (as shown in Table 4-3), 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. The minimum argument list size is 16 bytes. This space must be allocated in the frame for non-leaf procedures, but may remain unused.

The standard argument register use conventions are shown in Table 4-6. When making an indirect call, the floating point register arguments cannot be used. In this situation floating-point arguments must be passed in general registers as is shown in Table 4-6.

Function Return Values

Function result values are placed in registers as described in Table 4-8. As with value parameters, irregularly sized function results must be extended to 32 or 64 bits by right-justifying the value itself, and then left-extending it. Non-standard length function results that are signed integers are sign-extended left to 32 or 64 bits.

Table 4-6: Argument Register Use

	void	SID	nonFP	FP 32	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}

Table 4-7: Argument Register Use for Indirect Calls

	void	SID	nonFP	FP 32	FP64
arg word 0	no reg	sarg	arg0	arg0	arg1 {32..63}
arg word 1	no reg	arg1	arg1	arg1	arg1 {0..31}
arg word 2	no reg	arg2	arg2	arg2	arg3 {32..63}
arg word 3	no reg	arg3	arg3	arg3	arg3 {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)

When calling functions that return results larger than 64 bits, the caller passes a short pointer (using SR 4 - SR 7) in GR 28 (ret0) which describes the memory location for the function result. The caller is responsible for ensuring that the space for the result has been allocated. 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 4-8: Return Values.

Type of Return Value	Return Register
signed byte	ret0 (GR 28) - low order 8 bits
Nonfloating-Pt. (32-bit)	ret0 (GR 28)
Nonfloating-Pt. (64-bit)	ret0 (GR 28) - high order word
	ret1 (GR 29) - low order word
Floating-Pt. (32-bit)	fret (FPR 4)
Floating-Pt. (64-bit)	fret (FPR 4)
Space Identifier (32-bit)	sret (SR 1)
Any Larger Than 64-bit	result is stored to memory at location described by a short pointer passed by caller in GR 28[1]

1. The caller may not assume that the result's address is still in GR 28 on return from the function.

External Calls

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

In order for the object code in a dynamic 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, GR 19, as a pointer to the Linkage Table. The linker must generate import (calling) and export (called) stubs which set GR 19 to the Linkage Table pointer value for the target routine, and handle the inter-space calls needed to branch between dynamic 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 dynamic 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 dynamic 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.

Control Flow of an External Call

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

When building a dynamic library, the linker must generate import and export stubs for all procedures which can be called from outside of the dynamic library.

Figure 4-4 below shows the control flow of an external 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 GR 19 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

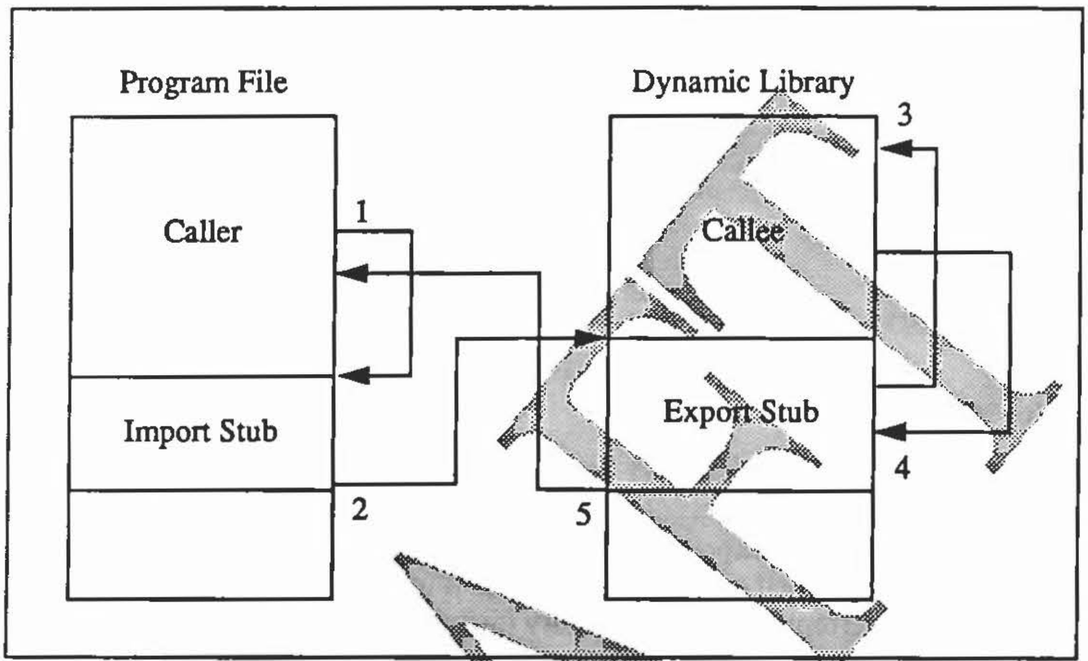

Figure 4-4: Flow of an External Procedure Call

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)
X':ADDIL    L'lt_ptr+ltoff,dp              ;load procedure entry point
   LDW      R'lt_ptr+ltoff(1),r21
   LDW      R'lt_ptr+ltoff+4(1),r19        ;load new GR 19 value.
   LDSID    (r21),r1                       ;load SID of proc. entry
   MTSP     r1,sr0                         ;move SID from GR to SR
   BE       0(sr0,r21)                     ;branch to target
   STW      rp,-24(sp)                     ;save RP'
```

The difference between a dynamic library and program file import stub is that the Linkage Table is accessed using GR 19 (the LTP) in a dynamic library, and is accessed using DP in the program file.

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

```
;Import Stub (Dynamic Library)
X':    ADDIL    L'ltoff,r19       ;load target (export stub) address
       LDW      R'ltoff(r1),r21
       LDW      R'ltoff+4(r1),r19 ;load new GR 19 (LTP) value
       LDSID    (r21),r1          ;load space id of target address
       MTSP     r1,sr0            ;move space id to space register
       BE       0(sr0,r21)        ;branch to target
       STW      rp,-24(sp)        ;save RP'
```

Called Code

The called code in dynamic 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 dynamic 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 dynamic 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:

```
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 SID from GR to SR.
       BE,N     0(sr0,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 GR 19 (the linkage table pointer register) must be stored at SP-32 by all PIC routines. This should be done once upon procedure entry. Register GR 19 must also be restored upon return from each procedure call, even if GR 19 is not referenced explicitly before the next procedure call. The LTP register, GR 19, 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 GR 19 into a callee-saves register as well, to avoid a memory reference when restoring GR 19 upon return from each procedure call. As with GR 27 (DP), the compilers must treat GR 19 as a reserved register whenever position-independent code is being generated.

Long Calls

Normally, the compilers generate a single-instruction call sequence using the BL instruction. The compilers will generate a long call sequence if the user explicitly requests long branch generation via a command-line option, or if the module is so large that the BL is not guaranteed to reach the beginning of the subspace (an offset of 240000 bytes leaving 22144 bytes for stubs). The existing long call sequence is three instructions, using an absolute target address:

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

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

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

Long Branches and Switch Tables

Long branches are similar to long calls, but are only two instructions because the return pointer is not needed:

```
        LDIL    L'target,%r1
        BE      R'target(%sr4,%r1)
```

For PIC, these two instructions must be transformed into four instructions, similar to the long call sequence:

```
        BL      .+8,%r1              ; get pc into r1
        ADDIL   L'target-L,%r1       ; add pc-relative offset
L:      LDO     R'target-L,%r1       ; add pc-relative offset
        BV,N    0(%r1)               ; and branch
```

The only problem with this sequence is when the long branch is to a switch table, where each switch table entry is restricted to two words. A long branch within a switch table must allocate a linkage table entry and make an indirect branch:

```
        LDW     T'target(%r19),%r1   ; load LT entry
        BV,n    0(%r1)               ; branch indirect
```

Here, the T' operator indicates request to the linker for a DLT_REL fixup (See "Fixup Requests" on page 10-30.)

Assigned GOTO Statements

ASSIGN statements in FORTRAN must be converted to a pc-relative form. For non-PIC code, the address is formed in a register then stored to the variable:

```
        LDIL    L'target,tmp
        LDO     R'target(tmp),tmp
```

This must be transformed into the following four-instruction sequence:

```
        BL      .+8,tmp              ; get rp into tmp
        DEPI    0,31,2,tmp           ; clear low-order bits
L:      ADDIL   L'target-L,tmp       ; get pc-relative offset
        LDO     R'target-L(%r1),tmp
```

Literal References

References to literals in the text space are handled exactly like ASSIGN statements (shown above). The LDO instruction can be replaced with LDW as appropriate.

Global and Static Variable References

References to global or static variables require two instructions either to form the address of a variable, or to load or store the contents of the variable:

```
; to form the address of a variable
ADDIL    L'var-$global$+x,%dp
LDO      R'var-$global$+x(%r1),tmp
; to load the contents of a variable
ADDIL    L'var-$global$+x,%dp
LDW      R'var-$global$+x(%r1),tmp
```

For position-independence, this sequence must be converted to use the linkage table pointer (GR19):

```
; to form the address of a variable
LDW      T'var(%r19),tmp1
LDO      x(tmp1),tmp2             ; omit if x==0
; to load the contents of the variable
LDW      T'var(%r19),tmp1
LDW      x(tmp1),tmp2
```

Note that the T' fixup on the LDW instruction allows for a 14-bit signed offset, which restricts the DLT to be 16Kb. Because GR19 points to the middle of the DLT, we can take advantage of both positive and negative offsets. The T' fixup specifier should generate a DLT_REL fixup proceeded by an FSEL override fixup. If the FSEL override fixup is not generated, the linker should assume that the fixup mode is LD/RD for DLT_REL fixups. In order to support DLT table sizes larger than 16Kb, the following long form of the above data reference must be generated.

```
; form the address of the variable
ADDIL    LT'var,%r19
LDW      RT'var(%r1),tmp1
LDO      x(tmp1),tmp2             ; omit if x==0
; load the contents of the variable
ADDIL    LT'var,%r19
LDW      RT'var(%r1),tmp1
LDW      x(tmp1),tmp2
```

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 4-5.

bits →	2	28	1	1
	SID	Address Part	L	X

Figure 4-5: Procedure Label Layout

The X field in the address section of the procedure label is reserved and must be set to 0. 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 dynamic libraries and incomplete executables is modified by the linker, when building dynamic 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 dynamic 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. The linker sets the L field (second-to-last bit) in the procedure label to flag this as a special PLT procedure label. The application must check this field to determine which type of procedure label has been passed, and call the target procedure accordingly. The X field is always 0 in a conforming application.

The following pseudo-code sequences show the process used to perform dynamic calls:

```
< dynamic library address load >
Clear L-field;
Load new LTP value into GR 19;
Load address of target;

< perform dynamic calls >
IF (L-field in Plabel) = 0 THEN
      Perform interspace branch using Plabel as target address;
ELSE BEGIN
      Perform < dynamic library address load >
      Perform interspace branch to target address;
END.
```

In order to generate a procedure label that can be used for dynamic 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.

Conforming compilers must generate the necessary code sequence required for proper handling of procedure labels. The code may be provided in a utility routine.

Stack Unwinding

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.

Overview

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.

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 PRO ABI. It is mandatory that the procedure entry and exit sequences conform to the standard specifications.

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

- The 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.

Fixup requests must be provided to the linker, providing the information to build the unwind descriptors. 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 table entries are four word entities with the format shown in Figure 4-6.

word #1	Starting address of the procedure
word #2	Ending address of the procedure
word #3	Unwind descriptor
word #4	

Figure 4-6: Format of an Unwind Table Entry

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.

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.

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.

The Unwind Descriptor

Using the information in the 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 the $UNWIND_START$ subspace. The definition of an unwind descriptor is shown in Figure 4-7.

```
struct unwind_table_entry {
    unsigned int region_start;                        /* Word 1 */
    unsigned int region_end;                          /* Word 2. */
    unsigned int Cannot_unwind:1;                     /* Word 3. */
    unsigned int Millicode:1;
    unsigned int Millicode_save_sr0:1;
    unsigned int Region_description:2;
    unsigned int reserved1:1;
    unsigned int Entry_SR:1;
    unsigned int Entry_FR:4;
    unsigned int Entry_GR:5;
    unsigned int Args_stored:1;
    unsigned int Variable_Frame:1;
    unsigned int Separate_Package_Body:1;
    unsigned int Frame_Extension_Millicode:1;
    unsigned int Stack_Overflow_Check:1;
    unsigned int Two_Instruction_SP_Increment:1;
    unsigned int Ada_Region:1;
    unsigned int reserved2:4;
    unsigned int Save_SP:1;
    unsigned int Save_RP:1;
    unsigned int Save_MRP_in_frame:1;
    unsigned int reserved3:1;
    unsigned int Cleanup_defined:1;
    unsigned int reserved4:1;                         /* Word 4 */
    unsigned int Interrupt_marker:1;
    unsigned int Large_frame_r3:1;
    unsigned int reserved5:2;
    unsigned int Total_frame_size:27;
};
```

Figure 4-7: Definition of an Unwind Descriptor

Unwind Descriptor Fields

region_start

This is the starting address of the unwind region.

region_end

This is the end address of the unwind region.

Cannot_unwind

One if this region does not follow unwind conventions and is therefore not unwindable; zero otherwise. (Non-unwindable code is discouraged.)

Millicode

One if this region is a millicode routine; zero otherwise.

Millicode_save_sr0

One if this (millicode) routine saves SR 0 in its frame (at current_SP - 16); zero otherwise.

Region_description

This field describes the code between the starting and ending offsets of this region. The values for this field are defined in Table 4-9.

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.

reserved1

This bit is reserved for future use. It must be set to 0 in conforming applications.

Table 4-9: Region_description Usage

Value	Context	Description
00	Normal	*Normal* context is code that falls between the last entry point and first exit point of a routine.
01	Entry point only	*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.
10	Exit point only	*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.
11	Discontinuous	*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.

Entry_SR

One if the sole entry-save space register SR 3 is saved/restored by the associated entry/exit code sequence; zero otherwise.

Entry_FR

The number of callee-save floating-point registers saved/restored by the associated entry/exit code sequence.

Entry_GR

The number of callee-save general registers saved/restored by the associated entry/exit code sequence. For example, a value of 5 in this field would mean that GR 3 through GR 7 (inclusive) have been saved.

Args_stored

One if this region's prologue includes storing any arguments to the routine in memory in the architected locations; zero otherwise. May be incorrect in optimized code.

Variable_Frame

Indicates that this region's frame may be expanded during the region's execution (using the Ada dynamic frame facility). Such frames require different unwinding techniques.

Separate_Package_Body

Indicates the associated region is an Ada separate package body. It has no frame of its own, but uses space in a parent frame to save RP and spill any entry save registers.

Frame_Extension_Millicode

Indicates the associated region is a special millicode routine which implements the Ada frame extension operation.

Stack_Overflow_Check

Indicates the associated region has an Ada stack overflow check in its entry sequence(s).

Two_Instruction_SP_Increment

Indicates the associated (Ada) region had a large frame such that two instructions were necessary to produce that portion of the frame increment which cannot be deduced from the frame size field in the unwind descriptor.

Ada_Region

One if the associated region should be treated as an Ada routine. This bit is intended for use by Ada exception handlers.

reserved2

These bits are reserved for future use. They must be set to 0 in conforming applications.

Save_SP

One if the entry value of SP is saved by this region's entry sequence in the current frame marker (current_SP - 4); zero otherwise.

Save_RP

For non-millicode, one if the entry value of RP is saved by the entry sequence in the previous frame (at previous_SP - 20); zero otherwise. For millicode, one if the entry values of MRP and sr0 are saved by the entry sequence in the current frame (at current_SP - 20 and current_SP - 16, respectively); zero otherwise. If this bit is one, the Save_MRP_in_frame and Millicode_save_sr0 bits are ignored.

Save_MRP_in_frame

One if the entry value of MRP is saved by the entry code in the current frame (at current_SP - 20); zero otherwise. Applies only to millicode.

reserved3

This bit is reserved and must be set to 0 in conforming applications.

Cleanup_defined

The interpretation of this field is dependent upon the language processor which compiled the routine.

reserved4

This bit is reserved and must be set to 0 in conforming applications.

Interrupt_marker

One if the frame layout corresponds to that of an interrupt marker.

Large_frame_r3

One if GR 3 is changed during the entry sequence to contain the address of the base of the (new) frame.

reserved5

This bit is reserved and must be set to 0 in conforming applications.

Total_frame_size

The amount of space, in 8-byte units, added to SP by the entry sequence of this region. This space includes register save and spill areas, as well as padding. This quantity is needed during unwinding to locate the callee-save register save area. It is also used to determine the value of previous_SP if it was not saved in the stack marker.

5

Object File and Library Formats

The Standard Object Module (SOM) format defined in this document is intended to be a common representation of code and data for PA-RISC based systems. A SOM may exist as a single entity or as part of a collection in a library.

SOM Format

The SOM consists of a SOM header record, an 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 word of the header record is also the first word of the SOM. It contains a *magic number* which distinguishes the SOM from any other entity. The header defines the size and location of the other components of the SOM.

The auxiliary header record was designed to be used by different systems in different ways. The Exec Auxiliary Header is a specific implementation for PRO ABI compliant systems. All other components are independent of the implementation.

The contents of the SOM may consist of the following components:

- SOM Header Record
- Auxiliary Header Area
- Space Dictionary
- Subspace Dictionary
- Space/Subspace Dictionary String Area
- Initialization Pointer Array
- Compilation Unit Dictionary
- Symbol Dictionary
- Fixup Request Array
- Compilation Unit/Symbol Dictionary String Area

A block diagram of the SOM format is shown in Figure 5-1.

Figure 5-2 shows a suggested layout of records in a SOM. The SOM header must be the first record in the SOM, and the Auxiliary header area must follow. The relative ordering of the remaining areas is not defined.

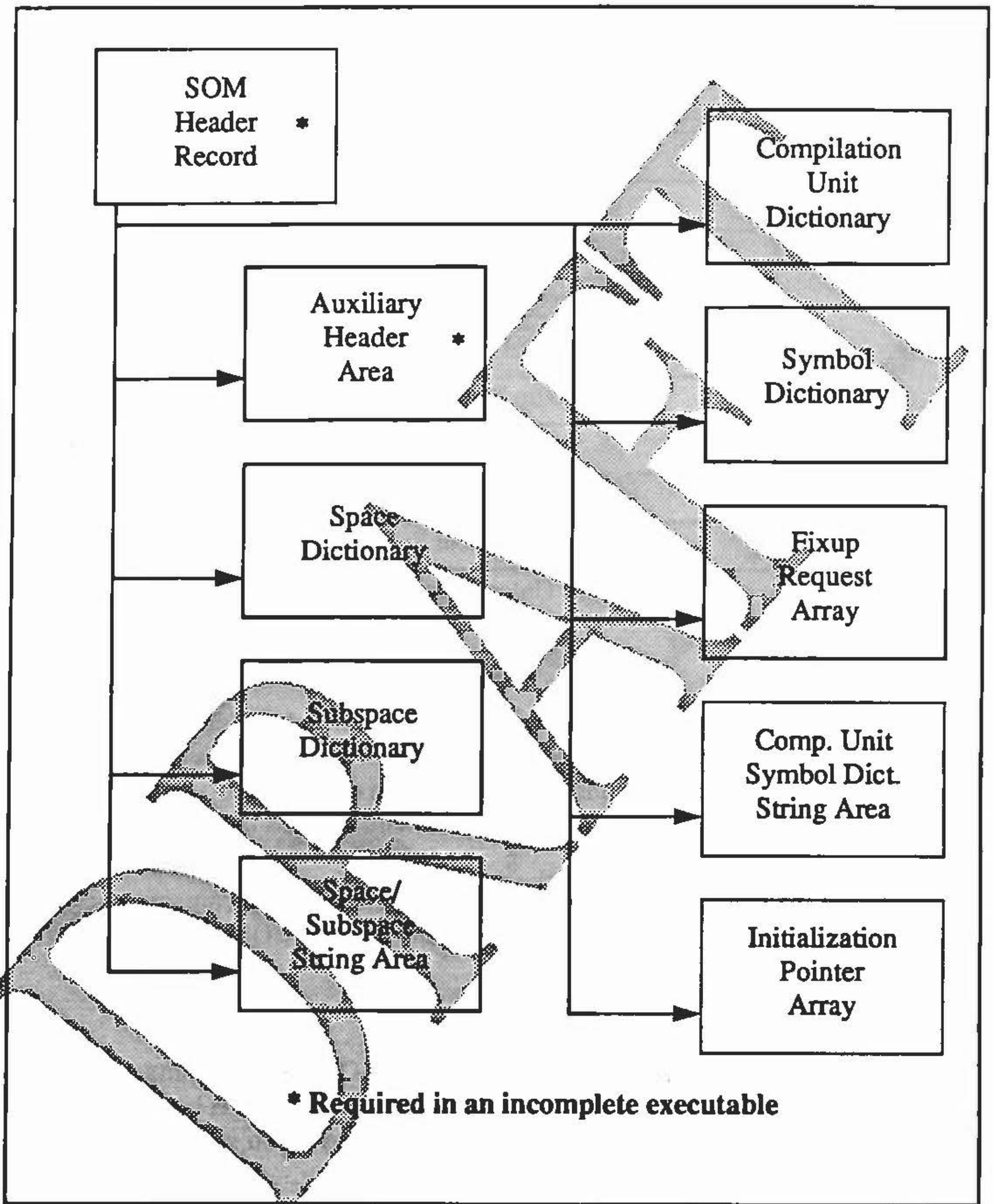

Figure 5-1: Block Diagram of the SOM Format

Figure 5-2: Suggested Record Layout of a SOM

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

SOM Header

The SOM header is a single record defining the location and size of all components of the SOM. The location of each component is given as a byte offset from the first byte of the header record, which is also the first byte of the SOM. The size of each component will be given as either the number of entries or as the total number of bytes.

In addition to the component identifications, the header record contains a version ID and the main entry point of the code represented by the SOM.

Auxiliary Header Area

The auxiliary header area is an optional entry in the SOM for containing additional information pertaining to the SOM. The information in this area is broken into one or more free format auxiliary headers, each identified by a two word auxiliary header identifier.

Space Dictionary

The space dictionary is an array of entries which define each space defined by the SOM. In addition to the space name, the entry record specifies the subspace dictionary entries, the space reference entries, and the initialization pointer entries for that space.

The space dictionary may contain definitions of spaces which are not loaded at run time. Subspaces within such spaces may contain information which must be carried along with the actual program, such as debugging and module interface information.

Subspace Dictionary

The subspace dictionary is an array of entries which define each subspace in each space of the SOM. Subspace entry records are grouped according to the space in which the subspace resides, and will contain information allowing the subspace to be relocated within the given space.

Each subspace record will contain the subspace name, access rights, alignment requirements, and fixup information. The subspace will be the smallest unit which may be relocated, and will also be the only unit to which access rights will be assigned.

String Areas

There are two string areas in the SOM. One area contains the strings used in the space and subspace dictionaries, and the other contains the strings used in the compilation unit and symbol dictionaries. Each string begins on a word boundary, and consists of a header word, specifying the total number of characters in the string, followed by the character data of the string. Each string is terminated by a byte containing zero, and any bytes residing between the zero byte and the word boundary are unused.

Initialization Pointers and Initialization Data Areas

The initialization pointer array contains an array of records which define contiguous areas in a space which are to be initialized at load time. A single initialization pointer can describe several subspaces.

Compilation Unit Dictionary

The Compilation Unit Dictionary consists of an array of entries which provide version identification for a SOM. Each SOM produced by a single compilation contains one compilation unit record. The compilation unit record may contain information about the language used to produce the SOM, compilation timestamps, product identification and copyright information.

Symbol Dictionary

The symbol dictionary consists of an array of entries defining each symbol in the SOM. There is one entry in the dictionary for each symbol that was either defined or referenced in the SOM. Each symbol record contains information about the type of the symbol, the scope of the symbol (i.e global or universal), and an index to the subspace that either defined or referenced the symbol defined.

Fixup Request Array

The fixup request array consists of an array of entries defining each fixup request in the SOM. Each entry specifies a location in a subspace which needs to be "fixed-up", or patched with some value not known at compilation time. For example, code addresses which are symbolically expressed by code labels in order to allow for relocation by the linker. Fixup records are grouped by subspace and contain symbol values and constants which are used to determine the fixup value.

SOM Object Files

The SOM header is a single record which defines the location and size of all other components of the SOM. The location of each component is given as a pointer relative to the first byte of the header record, and the size of each component is given either as the number of entries or as the total size in bytes of the component.

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.

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 5-3.

Note

This chapter specifies the SOM format as it is required for incomplete executables and dynamic libraries. Additional information covering functionality that is optional in these files is given in chapter 10.

```
struct header {
    short int       system_id;
    short int       a_magic;
    unsigned int    version_id;
    unsigned int    file_time[2];
    unsigned int    reserved1; entry-space
    unsigned int    reserved2; entry-subspace
    unsigned int    entry_offset;
    unsigned int    aux_header_location;
    unsigned int    aux_header_size;
    unsigned int    som_length;
    unsigned int    presumed_dp;
    unsigned int    space_location;
    unsigned int    space_total;
    unsigned int    subspace_location;
    unsigned int    subspace_total;
    unsigned int    loader_fixup_location;
    unsigned int    loader_fixup_total;
    unsigned int    space_strings_location;
    unsigned int    space_strings_size;
    unsigned int    init_array_location;
    unsigned int    init_array_size;
    unsigned int    compiler_location;
    unsigned int    compiler_total;
    unsigned int    symbol_location;
    unsigned int    symbol_total;
    unsigned int    fixup_request_location;
    unsigned int    fixup_request_total;
    unsigned int    symbol_strings_location;
    unsigned int    symbol_strings_size;
    unsigned int    unloadable_sp_location;
    unsigned int    unloadable_sp_size;
    unsigned int    checksum;
};
```

Figure 5-3: Definition of SOM Header Fields

SOM Header Fields

system_id

This field is used to identify the architecture that this object module is targeted for. The system ID for PA-RISC 1.1 systems is 210 (hexadecimal).

Binaries built for PA-RISC 1.0 systems have a system ID of 20B. PA-RISC 1.0 applications are supported by the PRO ABI, but conforming systems must support the PA-RISC 1.1 architecture.

a_magic

This is a number that indicates certain characteristics about the internal format of the object module. The magic numbers that are currently defined for use with SOMs on PA-RISC systems are listed in Table 5-1.

Table 5-1: Magic Number Values

Magic Number	SOM Type
0x106	Relocatable SOM
0x107	Non-sharable, executable SOM (EXEC_MAGIC)
0x108	Sharable, executable SOM (SHARE_MAGIC)
0x10B	Sharable, demand-loadable executable SOM (DEMAND_MAGIC)
0x10E	Dynamic Library

version_id

This field represents the version of the SOM format used, represented by the date the SOM version was defined. The only version_id that is currently defined for executables and dynamic libraries is 85082112. Relocatable objects use 87102412 (see chapter 10, "Relocatable Objects".) The version ID can be interpreted by viewing it in its decimal form and separating it into character packets of YYMMDDHH.

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.

reserved1 - reserved2

These fields are reserved by the PRO ABI for future use. They must be set to 0 in conforming applications.

entry_offset

This is the byte offset of the main entry point of the SOM relative to the first byte of the space. The two low-order bits specify the minimum privilege level required.

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 Header Area" on page 5-4. 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 dynamic libraries. It contains the value of the data pointer (DP) assumed during compilation or linking of this SOM. In a dynamic 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. Setting all bits to zero in *loader_fixup_location* indicates that the loader fixup array is not defined in a SOM. The loader fixup array must start on a word boundary. *Loader_fixup_location* is not used by the PRO ABI, and its value is undefined.

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* means that the loader fixup array is not defined in a SOM. *Loader_fixup_total* is not used by the PRO ABI, and its value must be set to 0.

space_strings_location

This field points to a string area that contains both space and subspace names. It is a byte offset relative to the first byte of the SOM header. Setting all bits to zero indicates that this area is not defined. The string area must start on a word boundary with 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. The initialization pointer array must start on a word boundary. *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 init pointer array. Setting all bits to zero in *init_array_total* means that the initialization pointer array is not defined in a 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 bytes 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

This field 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 and 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 start of the SOM header, of the first byte of the data for unloadable spaces. Setting all bits to zero indicates that there are no unloadable spaces defined in a SOM. The data for unloadable spaces must be double-word aligned and 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.

Auxiliary Header Areas

A SOM object module or a SOM library file may optionally contain an auxiliary area to hold information specific to a particular implementation or application. The location and size of the auxiliary header area is specified in the SOM or LST header. (See "LST Header" on page 10-45.)

If the auxiliary area is present it will contain one or more auxiliary header records. The first two words of every auxiliary header record 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.

A conforming application may not contain an implementation-specific header. This type is listed in the following section only to reserve the use of the type.

The structure of the auxiliary header id defined below in Figure 5-4.

```
struct aux_id {
   unsigned int  undefined:16;
   unsigned int  type:16;
   unsigned int  length;
};
```

Figure 5-4: Definition of the Auxiliary Header

Auxiliary Header Fields

undefined

These bits are not defined by the ABI. They are reserved for application use and must be ignored by the loader.

type

This field is a numeric value that defines the contents of the auxiliary header. Values less than or equal to 32767 are reserved for PRO defined auxiliary header record types. TYPE values greater than 32767 are user definable. The currently defined auxiliary header type values are listed in Table 5-2.

Table 5-2: Auxiliary Header Types

Value	Usage
1	Linker auxiliary header
3	Debug auxiliary header
4	Exec auxiliary Header
6	Version strings
9	Copyright auxiliary header
10	Dynamic library version information
11	Implementation-specific header

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, but the next auxiliary header or the end of the auxiliary header area will be word aligned. 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.

Exec 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 dynamic libraries.

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

Figure 5-5: Definition of Exec Auxiliary Header

Exec Auxiliary Header Fields

som_auxheader

This field contains the auxiliary header identifier for an exec auxiliary header. See "Auxiliary Header Fields" on page 5-17.

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. The value of *exec_tsize* must be in the range (0x0 ≤ *exec_tsize* < 0x40000000).

exec_tmem

This field specifies the space-relative byte offset of text (code) in memory. The address must be page aligned. The value of *exec_tmem* must be in the range (0x1000 ≤ *exec_tmem* < 0x40000000).

Note

The total value of *exec_tsize* + *exec_tmem* must be less than 0x40000000 bytes.

---

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 and must be aligned on a page boundary.

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. The value must be in the range (0x0 ≤ *exec_dsize* < 0x40000000) for SHARE_MAGIC and DEMAND_MAGIC files. In EXEC_MAGIC files, the upper limit is 0x70000000.

exec_dmem

This field specifies the space-relative byte offset of data in memory. The address must be 4 Kbyte aligned. For SHARE_MAGIC and DEMAND_MAGIC files, the value must be in the range (0x40001000 ≤ *exec_dmem* < 0x80000000). In EXEC_MAGIC files, the lower bound is the address of the first full page beyond text (*exec_tmem* + *exec_tsize*)

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 and must be page-aligned.

exec_bsize

This field contains the size in bytes of the uninitialized data in the file. The value must be in the range (0x0 ≤ *exec_bsize* < 0x40000000) for SHARE_MAGIC and DEMAND_MAGIC files. In EXEC_MAGIC files, the upper limit is 0x70000000.

exec_entry

This field contains the space-relative byte offset of the main entry point for this file. The value is restricted to (*exec_tmem* ≤ *exec_entry* < (*exec_tmem* + *exec_tsize*)).

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.

The remaining bits are reserved for future use.

exec_bfill

This field specifies the value to which uninitialized data (BSS) should be initialized. Conforming applications must set this field to 0, but this field may be ignored by conforming systems.

Linker Footprint Auxiliary Header

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. The linker footprint auxiliary header is shown in Figure 5-6.

```
struct linker_footprint {
   struct aux_id  header_id;
   char  product_id[12];
   char  version_id[12];
   int  htime[2];
};
```

Figure 5-6: Definition of Linker Footprint Auxiliary Header

header_id

This is the auxiliary header id for the linker footprint.

product_id

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

version_id

This eight character array contains the 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. It has the same form as the SOM header *file_time* field.
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.

Debugger Footprint Auxiliary Header

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. The debugger footprint auxiliary header is shown below in Figure 5-7.

```
struct debugger_footprint {
   struct aux_id header_id;
   char debugger_product_id[12];
   char debugger_version_id[8];
   unsigned int debug_time[2];
};
```

Figure 5-7: Definition of a Debugger Footprint Auxiliary Header

header_id

This is the auxiliary header id for the debugger footprint aux header.

debugger_product_id

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

debugger_version_id

This eight character array contains the 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. It has the form of the SOM header *file_time* field 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.

Copyright Auxiliary Header

The copyright auxiliary header is used to embed a copyright string in an application. 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 (\0) terminator. (Note that the length field in *aux_header_id* includes both the *string_length* field and the padding bytes of the string.)

```
struct copyright_aux_hdr {
   struct aux_id  header_id;
   unsigned int  string_length;
   char  copyright[1];
};
```

Figure 5-8: Definition of the Copyright Auxiliary Header

Version String Auxiliary Header

The version string auxiliary header can be used for any user-defined version 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 (\0) terminator. (Note that the length field in *aux_header_id* includes both the *string_length* field and the padding bytes of the string.)

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

Figure 5-9: Definition of User String Auxiliary Header

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

The string is defined by the character data given here.

Initialization Pointers

The initialization pointer array is used to determine how to initialize virtual space when a file is loaded. The fields in the initialization pointer record are very similar to the fields in the subspace record, but the initialization pointer record can be used to initialize more than one subspace. The initialization pointer information is used by the loader after subspaces have been relocated and are in their "final" position within a space. Compilers should use the fields provided in the subspace record to convey initialization information to the linker, since relocatable subspaces are not guaranteed to remain contiguous.

Initialization pointers only occur in executables and dynamic libraries. While they are defined for conforming applications, conforming systems are not required to make use of this functionality.

```
struct init_pointer_record {
   unsigned int  space_index;
   unsigned int  access_control_bits:7;
   unsigned int  has_data:1;
   unsigned int  memory_resident:1;
   unsigned int  initially_frozen:1;
   unsigned int  new_locality:1;
   unsigned int  reserved:21;
   unsigned int  file_loc_init_value;
   unsigned int  initialization_length;
   unsigned int  space_offset;
};
```

Figure 5-10: Definition of an Initialization Pointer Record

Initialization Pointer Fields

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:

offset = *space_index* * size of (space record)
+ *space_dictionary_location* (found in the SOM header)
+ address of the first byte of the SOM header.

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

access_control_bits

See "access_control_bits" on page 10-11.

has_data

If this flag is set to one, then data is defined in the SOM for this subspace.

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. Conforming applications must use a value of 0.

initially_frozen

If this flag is set to one then the subspace is to be locked in physical memory when the operating system is being booted. Conforming applications must set this flag to 0.

new_locality

This flag indicates that this initialization pointer begins a new locality set. The loader can use this bit to determine which initialization pointers belong to each locality set, so that it can swap entire locality sets together.

reserved

These bits are reserved for future expansion. Conforming applications must set these bits to zero.

file_loc_init_value

If the *has_data* bit is set, this field contains a byte offset relative to the first byte of the SOM header. The field *file_loc_init_value* points to the data used to initialize one or more subspaces.

If *has_data* is zero then this field contains a 32 bit quantity which is used as an initialization pattern for the subspace(s).

initialization_length

If the *has_data* flag is set then this field contains the size in bytes of the initialization area in the file. If the flag *has_data* is set to zero, then this field indicates the size of the area that the loader must fill with the bit pattern contained in the field *file_loc_init_value*.

space_offset

This is a byte address relative to the beginning of a space where initialization is to start.

Dynamic Library File Definition

A dynamic library (sometimes referred to as *shared libraries*, see glossary for definitions) contains subroutines that are attached to the program at run time. Program files are smaller since they do not contain private copies of library routines, and updates to a dynamic library take effect without needing to relink existing applications.

The DL header appears in every dynamic library and in incomplete executables (program files linked with dynamic libraries--may contain unsatisfied symbols which will be satisfied at run time by the dynamic loader). It must be at the location indicated by *exec_tmem* in the exec auxiliary header (and at the location indicated by exec_tfile in the disk image). It defines fields used by the dynamic loader and various other tools when attaching the dynamic 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.

The structure of a dynamic library is shown in Figure 5-11.

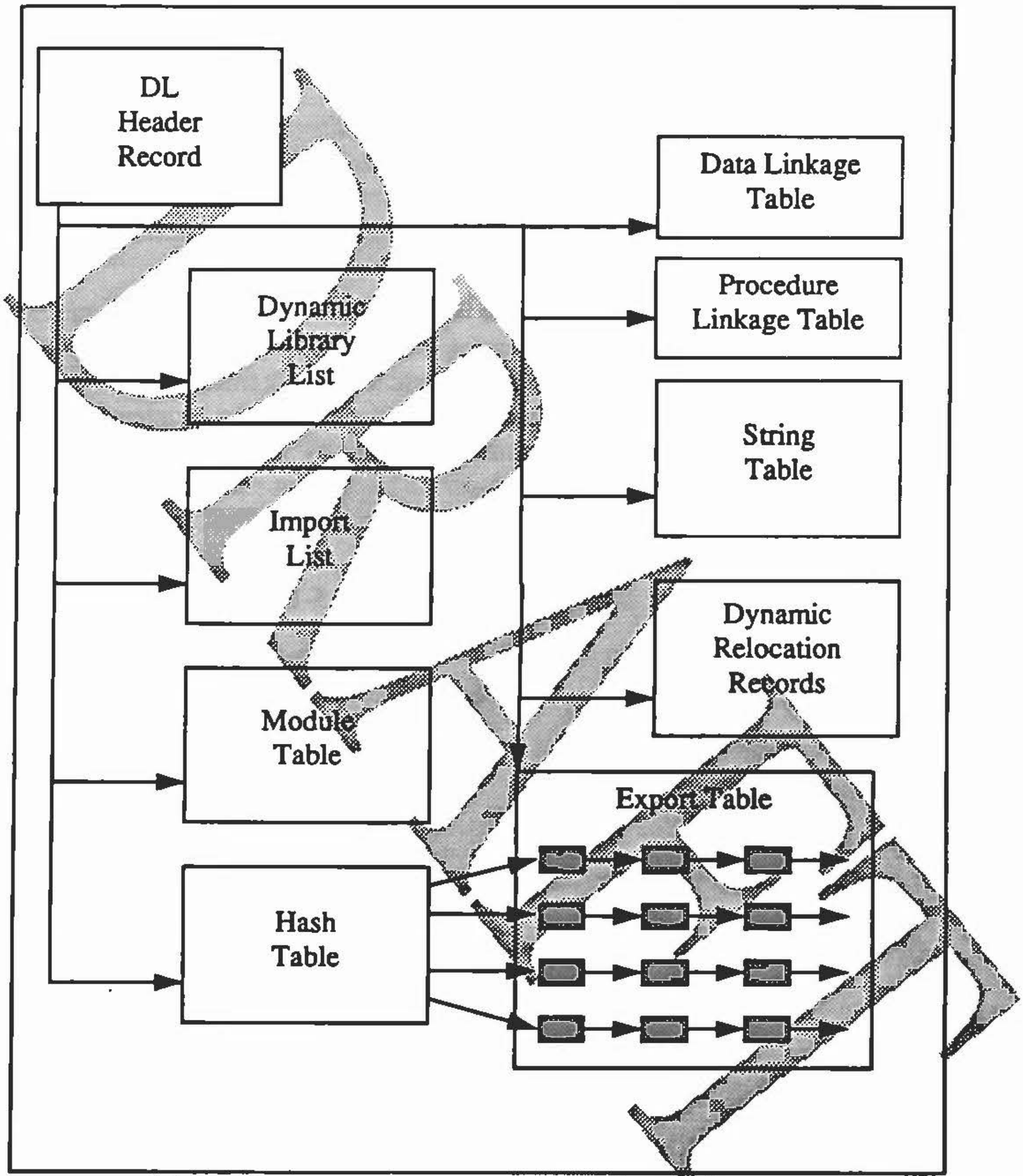

Figure 5-11: Structure of Dynamic Library Header

```
struct dl_header {
    int      hdr_version;        /* header version number */
    int      ltptr_value;        /* data offset of LT ptr (R19) */
    int      shlib_list_loc;     /* text offset of shlib list */
    int      shlib_list_count;   /* # of items in shlib list */
    int      import_list_loc;    /* text offset of import list */
    int      import_list_count;/* # of items in import list */
    int      hash_table_loc;     /* text offset of hash table */
    int      hash_table_size;    /* # of slots in hash tbl */
    int      export_list_loc;    /* text offset of export list */
    int      export_list_count;/* # of items in export list */
    int      string_table_loc; /* text offset of string table */
    int      string_table_size;/* length of string table */
    int      dreloc_loc;         /* text offset of dreloc recs */
    int      dreloc_count;       /* # of dynamic relocation recs */
    int      dlt_loc;            /* data offset of DLT */
    int      plt_loc;            /* data offset of PLT */
    int      dlt_count;          /* # of DLT entries in LT */
    int      plt_count;          /* # of PLT entries in LT */
    short    highwater_mark;     /* highest version # in lib */
    short    flags;              /* various flags */
    int      export_ext_loc;     /* text offset of EET */
    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;      /* string table index to path */
                                 /* (must be > 0 to be valid) */
    int      initializer_count;/* # of initializers declared */
    int      reserved1;          /* reserved, initialized to 0 */
    int      reserved2;          /* reserved, initialized to 0 */
};
```

Figure 5-12: Definition of DL Header

DL Header Fields

hdr_version

This field is used to denote the version of the DL header. It is currently set to the decimal number "89060912".

ltptr_value

This field is the data-relative offset of the Linkage Table pointer (GR 19 for dynamic 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 dynamic libraries. All data references and PIC code in a dynamic library must go indirectly though the linkage pointer.

shlib_list_loc

This field is the text-relative offset of the dynamic library list. The dynamic library list is a list of dynamic libraries that the given file depends on for symbol bindings. If the dynamic library list in a dynamic library is present, the dynamic library is said to "depend" on the libraries in the dynamic library list. See "Dynamic Library List" on page 5-42.

shlib_list_count

This field is the number of entries in the dynamic 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

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. See "Export Hash Table" on page 5-49.

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 dynamic library 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 PLT contains entries for each unresolved procedure call in a dynamic 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

The highest version number in the dynamic library. 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 dynamic library symbol is to be used for binding the program file's symbol reference.

flags

These flag bits are used to identify the search path used to resolve dynamic libraries at run time. Libraries in the dynamic library list may specify that these path lists are used rather than the absolute name recorded at link time. See "Dynamic Library List" on page 5-42.

Table 5-3: DL Header Flag Types

Flag	Value (hex)	Definition
ELAB_DEFINED	0x1	The *elaborator* field is defined.
INIT_DEFINED	0x2	The *initializer* filed is defined.
SHLIB_PATH_ENABLE	0x4	Search for the library using the SHLIB_PATH environment variable
EMBED_PATH_ENABLE	0x8	Search for the library using the path list referenced by *embedded_path*
SHLIB_PATH_FIRST	0x10	If both mechanisms are available, the path referenced by SHLIB_PATH is searched first if this bit is set.

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 dreloc 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 dynamic library. It contains 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_DEFINED bit in the *flags* field is set.

initializer

This field holds an index into the import table if the INIT_DEFINED bit in the *flags* field is set and the *initializer_count* field is zero. If *initializer_count* is non-zero, then this field is the address of a word-aligned array of text-relative indices into the import list for each initializer.

embedded_path

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

initializer_count

The number of entries in the initializer import list.

reserved1 - reserved2

These fields are reserved for future use (currently set to 0).

Dynamic Relocation

Dynamic relocation records (loader fixups) are used by the dynamic loader to apply run-time patches to the data area of dynamic libraries and incomplete executables. Dynamic relocation records are built for each dynamic library data item initialized to the address of a function or variable. These records are needed since the final addresses of the library code and data are not known until run time.

```
struct dreloc_record {
   int          object_type;    /* incomplete executable or dl */
   int          symbol;         /* index into import table */
   int          location;       /* offset to patch (data-rel) */
   int          value;          /* text/data offset for patch */
   unsigned char  type;         /* type of dreloc record */
   char         reserved;       /* reserved */
   short        module_index;   /* index into module table */
};
```

Figure 5-13: Dynamic Relocation Record Definition

Dynamic Relocation Record Fields

object_type

Initialized to (-1) if a dynamic library, 0 if an incomplete executable.

symbol

The *symbol* field is an index into the import table if the relocation is an external type.

location

The *location* field is the data-relative offset of the data item the dreloc record refers to.

value

This is the text or data-relative offset to use for a patch if it is an internal fixup type.

type

This field represents the type of the dynamic relocation record. Valid relocation types are:

```
#define DR_PLABEL_EXT   1  /* initialized to an external code plabel */
#define DR_PLABEL_INT   2  /* initialized to internal code plabel*/
#define DR_DATA_EXT     3  /* initialized to an external data symbol */
#define DR_DATA_INT     4  /* initialized to an internal data offset */
#define DR_TEXT_INT     7  /* initialized to an internal text offset */
```

The only valid type in a program file is:

```
#define DR_PROPAGATE    5  /* data item from a dynamic library */
```

reserved

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

module_index

This field is an index into the module_table of the module associated with this relocation record. For DR_PROPAGATE dreloc records, this field is set to (-1).

Linkage Table

The Linkage Table is located in the $DATA$ space of a dynamic library and/or program file. It is divided into two parts: a Data Linkage Table for data references and a Procedure Linkage Table for procedure calls. The linkage table is used as a branch table to handle indirect procedure and data references.

Data Linkage Table

The Data Linkage Table contains a one-word entry for each data item that is referenced by PIC. It is initialized by the dynamic loader once the addresses of all dynamic library data are known. This table is usually only in dynamic libraries.

Procedure Linkage Table

The Procedure Linkage Table may reside in a program file and/or dynamic library. In a dynamic library, it contains entries for each unresolved procedure call and/or each call to an exported procedure symbol. In a program file, it contains entries for each dynamic library procedure call.

The PLT is initialized by the dynamic loader.

```
struct PLT_entry {
   int      proc_addr;  /* address of procedure */
   int      ltptr_value;/* value of GR 19 reg for this proc */
};
```

Figure 5-14: PLT Entry Definition

Procedure Linkage Table Fields

proc_addr

This field contains the address of the procedure to be branched to, taken from the export table of a dynamic 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, and 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.

Dynamic Library List

The dynamic library list is built for both dynamic libraries and incomplete executables. It is allocated in the $TEXT$ space. The list contains an entry for each dynamic library specified at static link time.

The first entry will be the absolute path name at link time of the executable or library that contains the dynamic library list.

```
struct shlib_list_entry {
    int  shlib_name;        /* offset into string table */
    unsigned char  dash_l_reference;  /* used for lib spec */
    unsigned char  bind;  /* bind immediate, deferred */
    short  highwater_mark; /* highest exported sym version */
};
```

Figure 5-15: Dynamic Library List Entry Definition

Dynamic Library List Fields

shlib_name

This field contains an index into the dynamic library string table of the absolute path name of the dynamic library specified at static link time.

dash_l_reference

This field is a flag to denote if the absolute path name of the dynamic library specified at link time is to be used at run time. If this flag is set to FALSE, the dynamic loader will search for those libraries specified at link time using the path(s) given. If dash_l_reference is TRUE, the absolute path name is expected at runtime. This allows a different path to be searched at run time than what was specified at link time. See “flags” on page 5-34.

bind

This field describes the binding-time preference specified at link time when the program is built. The binding modes are defined in Table 5-4.

highwater_mark

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

Table 5-4: Defined Binding Methods

Binding Flag	Value	Description
BIND_IMMEDIATE	0x00	Symbols are bound at program start-up.
BIND_DEFERRED	0x01	Symbols are bound at first reference.
BIND_FIRST	0x04	Insert the loaded library before all others in the current link order.
BIND_NONFATAL	0x08	Allow binding of unresolved symbols.
BIND_NOSTART	0x10	Causes the dynamic loader to not call the initializer
BIND_VERBOSE	0x20	Make dynamic loader display verbose messages when binding symbols.
BIND_RESTRICTED	0x40	Causes the search for a symbol definition to be restricted to those symbols that were visible when the library was loaded.

Dynamic Library Import List

The import list is created for both incomplete executables and dynamic libraries. It resides in the $TEXT$ space of the object and is made up of an array of import entries. Each entry is a one-to-one correspondence with the linkage table. There is an import symbol for each DLT entry in the linkage table and each PLT entry in the linkage table. Import entries of type ST_NULL are used to preserve this correspondence for internally resolved linkage table entries.

```
struct import_entry {
   int       name;          /* offset into string table */
   short     reserved1;     /* Currently unused */
   unsigned char  type;     /* symbol type */
   unsigned int  bypassable:1;
   unsigned int  reserved2:7;
};
```

Figure 5-16: Dynamic Library Import List Record Definition

Dynamic Library Import List Fields

name

This field contains an offset into the string table denoting the symbol name. If no symbol name is defined, this field is set to (-1).

reserved1

This symbol is reserved for future use. It must be initialized to (-1) in conforming applications.

type

This field specifies the symbol type. Valid symbol types are ST_NULL, ST_CODE, ST_DATA, ST_STORAGE, and ST_PLABEL. See Table 10-3, on page 10-21 for the definitions of these types.

bypassable

This bit is set to 1 if the import list is contained in a dynamic library, and the import is a code symbol that does not have its address taken in this dynamic library.

reserved2

These bits are reserved for future expansion. They must be initialized to 0.

Dynamic Library Exports

The export table is a hashed table of the symbols exported by a dynamic library or incomplete executable. The dynamic loader uses this table to look up symbols at run time when establishing linkages from programs and other libraries.

The structure of an export table entry is shown in Figure 5-17 and the hashing function used with the export table is shown in Figure 5-18.

```
struct export_entry {
   int   next;
   int   name;
   int   value;
   union {
      int  size;
      struct misc_info  misc;
   } info;
   unsigned char  type;
   char  reserved1;
   short module_index;
};

struct misc_info {
   short version;
   unsigned int  reserved:6;
   unsigned int  arg_reloc:10;
};
```

Figure 5-17: Dynamic Library Export Entry Definition

Export Entry Fields

next

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

name

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

value

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

info

If the exported symbol is of type ST_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

This field specifies the symbol type. Valid symbol types are ST_CODE, ST_DATA, ST_STORAGE, and ST_PLABEL. See Table 10-3, on page 10-21 for the definitions of these types.

reserved1

These bits are reserved for future expansion and must be initialized to 0.

module_index

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

Export Hash Table

The export hash table is used to rapidly locate entries in the export table. Each entry is hashed using the function in Figure 5-18. The index is calculated by the value of the *hash_string* function modulo the number of hash slots (exported symbols). The hash table itself consists of one word entries. Each entry is the head of a linked list of export entries that hash to the same location.

The entries in each list are specified by their index in the export table, with a value of (-1) terminating the list. These lists are not required to be ordered in any manner.

s

```
unsigned int hash_string(s)
register unsigned char *s;
{
   register unsigned int key;

   key = 0;
   while (*s)
      key = ((key << 5) | (key >> 27)) ^ *s++;
   return (key);
}
```

Figure 5-18: Hash Algorithm used with Export Table

Dynamic Library Export Entry Extension

The export entry extension table is located in the $TEXT$ space of the object in a dynamic library. There is a one-to-one correspondence between the export table and export extension table. This table contains information needed for data copying from a dynamic library file to an incomplete executable.

```
struct export_entry_ext {
    int size;
    int dreloc;
    int same_list;
    int reserved1;
    int reserved2;
};
```

Figure 5-19: Dynamic Library Export List Record Definition

Export Entry Extension Fields

size

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

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

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.

reserved1

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

reserved2

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

Dynamic Library Module Table

The module table is located in the $TEXT$ space of a dynamic library. It provides information on the modules that make up the dynamic library.

The dynamic library module table is optional.

```
struct module_entry {
   int    drelocs;
   int    imports;
   int    import_count;
   char   flags;
   unsigned short  module_dependencies;
   int    reserved;
}
```

Figure 5-20: Definition of a Module Table Entry

Dynamic Library Module Table Fields

drelocs

This field is a text address (subject to relocation) of a table of one word indices into the dynamic relocation table. This table is terminated by an entry with the value (-1).

A value of (-1) for *drelocs* indicates that no dynamic relocation entries are associated with this table.

imports

This field contains a text address (subject to relocation) of a table of one-word indices. The table lists *module_dependencies* indices into the module table, followed by *import_count* indices into the import list. The modules and symbols indicated must be resolved before the module can be used.

A value of (-1) indicates that this array does not exist.

import_count

This field is the number of import symbol entries in the import list required by this module.

flags

The value ELAB_REF (0x1) is used here to indicated that an elaborator was referenced in the module.

module_dependencies

This field is the number of modules that the current module must bind before all of its own import symbols can be bound.

reserved

This field is reserved for future use and must be initialized to 0.

6

Program Loading and Dynamic Linking

While most of the details regarding system implementation are hidden from applications, the requirements for invoking a program must be specified. The system must be able to pass arguments to the application, provide access to the system functionality, and begin execution of the application.

Executable Binary File Format

Both chapter 5, Object File and Library Formats, and chapter 10, Relocatable Objects, define the SOM file format in detail. In this section, the specific requirements of an incomplete executable are discussed.

An executable binary file must contain a SOM header, exec auxiliary header, text and data sections. An uninitialized data section, or BSS, may optionally be included. Additional sections may exist within the executable binary, but the loader must not be expected to interpret them.

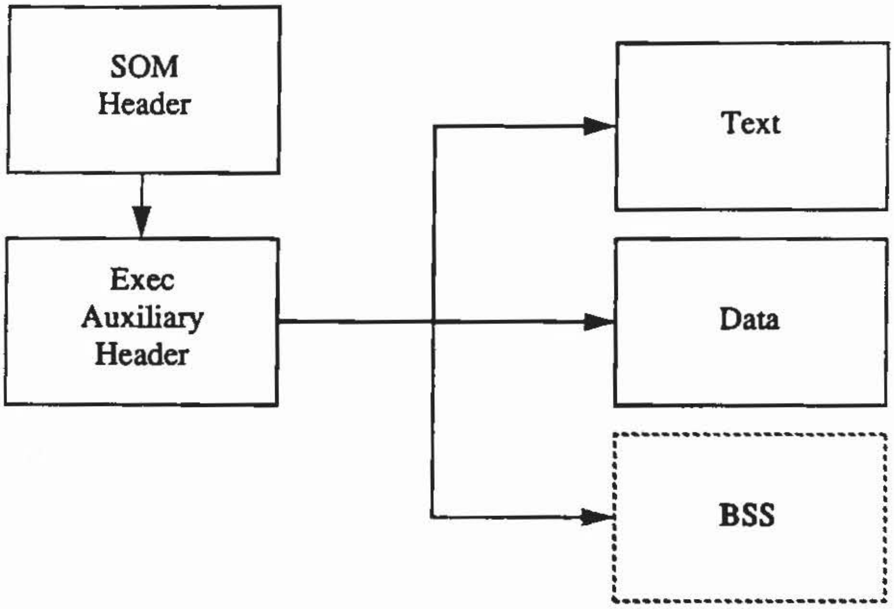

Figure 6-1: Executable Binary File Format

The SOM header must appear first in the binary, followed immediately by the exec auxiliary header. These headers are described in chapter 5. The exec auxiliary header specifies the location and size of the text, data and BSS sections.

Text

The text section contains the instructions for the conforming application. The text section must be aligned on a page boundary in both the file and in memory. The text section must be contained in the first quadrant.

Although the size of the text section indicated by the *exec_tsize* field does not need to be a multiple of 4 Kbytes, the actual size of the text section in the file must be a multiple of 4 Kbytes. This may be accomplished by padding the text section with zeroes.

The dynamic library header must be at the start of the text area.

Data

The data section contains the initialized data for the conforming application. The data section must start at an offset from the start of the executable binary file that is a multiple of 4 Kbytes. It must also be aligned on a page boundary when loaded into memory. The data section must be contained in the second quadrant.

Although the size of the data section indicated by the *exec_dsize* field of the exec auxiliary header does not need to be a multiple of 4 Kbytes, the actual size of the data section must be a multiple of 4 Kbytes. The data section can be padded with zeroes to make it a multiple of 4 Kbytes.

BSS

The BSS section describes the uninitialized data for the conforming application. The BSS begins at the first page-aligned address past the data segment. The size of the BSS section is defined by the *exec_bsize* field of the exec auxiliary header.

While the BSS section is said to contain uninitialized data, it is actually required to be initialized with 0's.

Program Loading

All programs must include start-up code. This code defines entry points, initializes program variables and checks for dynamic libraries. The symbols defined by the initialization code are listed in Table 6-1. An example implementation is shown in Figure 6-2 and Figure 6-2. This version of a start-up routine is built from both PA-RISC assembly and C language source.

Execution begins at $START$. The global data pointer (DP/GR 27) and the stack pointer (SP/GR 30) are initialized. If a valid dl_header is found, then the dynamic loader (/usr/shlib/dld.sl) is called to initialize the linkage table and program data.

Note

There is no API-defined entry point into dld.sl. As the example code shows, the entry point for /usr/shlib/dld.sl is calculated from the SOM header and Exec Auxiliary header.

After the call to __map_dld, dynamic references are supported. For C-language programs, the call to _start will refer to a system-specific routine in /usr/shlib/libc.sl that calls main and terminates using _exit. For other languages, _start must be defined in the program text and result in the execution of the programs outer block.

A small piece of code, _sr4export, is also included in the start-up code. This code is used to support interspace calls.

The start-up code also identifies the top stack frame by setting the *Previous_SP* value (SP-4) to zero, with the *Save_SP* bit set in the unwind descriptor.

Table 6-1: Symbols Defined in a Conforming Executable

Symbol	Description
`__argc_value`	A variable of type int containing the argument count.
`__argv_value`	An array of character pointers to the arguments themselves.
`_environ`	An array of character pointers to the environment in which the program will run. This array is terminated by a null pointer.
`_SYSTEM_ID`	A variable of type int containing the system id value for an executable program.
`$START$`	Default execution start address.
`_start`	A secondary start-up routine for C programs, called from $START$, which in turn calls main. This routine is contained in the C library rather than in the crt0.o file. For Pascal and FORTRAN programs, this symbol labels the beginning of the outer block (main program) and is generated by the compilers.
`$global$`	The initial address of the program's data pointer. The start-up code loads this address into GR 27.
`__text_start`	The beginning address of the program's text area.[1]
`__data_start`	The beginning address of the program's data area.[1]
`_end`	First address above the uninitialized data region.[1]
`_etext`	First address above the program text.[1]
`_edata`	First address above the initialize data region.[1]

1. The symbols __text_start, __data_start, _end, _etext and _edata are defined by the linker.

Process Initialization

When the system activates a conforming application using exec, the register state must be set as defined in Table 6-2.

Additionally, the following apply:

- The value of all other general registers is unspecified.
- Space registers 0, 1, 2, and 3 are unspecified.
- The value of the Shift Amount Register (SAR) is unspecified. The value of the Interval Timer is unspecified and must not be read by a conforming application. The other control registers cannot be read from or written to by a conforming application.
- The values in all coprocessor registers (including floating-point registers) are undefined.
- If any of the bits in the Coprocessor Configuration Register (CCR) are set, then the corresponding coprocessors must be present and functional.
- The Processor Status Word (PSW) has the C, D, P and Q bits set to 1, and the B, M and N bits cleared to 0. The remaining bits are established at the discretion of the system.

In the case of the floating-point status register (FPR 0), the following conditions must apply (although a conforming application may set these bits during execution of its start-up code):

- All exception flags must be clear.
- All exception traps must be disabled.
- The rounding mode bits must be set to 0 (round to nearest).

Table 6-2: Register Usage at Process Initialization

Register	C Source Definition	Usage
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
SR 4		address of first quadrant of virtual address space[1]
SR 5		address of second quadrant of virtual address space[2]
SR 6		address of third quadrant of virtual address space[2]
SR 7		address of fourth quadrant of virtual address space

1. Space register 4 is unprivileged, but it must not be modified by a conforming application.
2. Space registers 5, 6 and 7 are privileged and cannot be modified by a conforming application.

```
DL_HDR_VERSION_ID .equ 89060912

 .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

$START$                        ; Entry point used for linker

 ldil L'$global$,dp           ; Initialize the global data
 ldo R'$global$(dp),dp        ; pointer
 ldo 128(sp),sp               ; Allocate frame, marker, & arg
 depi 0,31,3,sp               ; list and doubleword align sp

 ; 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
 ldw R'__text_start(r1),r31    ; dl_header version # (080189)
 addil L' DL_HDR_VERSION_ID, 0; offset 0x10
 ldo R' DL_HDR_VERSION_ID (1),19 ; offset 0x14
 combf,=,n 19,31,L$0001          ; offset 0x18

 .import __map_dld
 ; map_dld
 ; set sp to skip 64K to maintain clean stack (dld uses
 ; sp+64k for sp)
 copy sp, 7 ; save sp
 addil L'65536, sp
 ldo R'65536(1),sp
 addil L'__dld_sp-$global$(27)
 stw sp, R'__dld_sp-$global$(1)
 copy 26, 4                      ; save argc, argv and envp
```


Figure 6-2: Example Implementation of Start-up Code (assembly section)

```
 copy 25, 5
 copy 24, 6
 bl __map_dld,rp
 copy r3, arg0          ; pass in program file name

 copy 4, 26             ; restore argc, argv and envp
 copy 5, 25
 copy 6, 24
 copy 7, sp             ; restore original sp.

L$0001

 .import _start
 .call

 bl _start,rp
 stw r0,-4(sp)          ; mark last stack frame (null fm_psp)

 .proc                  ; _sr4export serves as target of calls
 .callinfo              ; 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

 .space $PRIVATE$
 .subspa $GLOBAL$
 .export $global$
 .WORD 0                ; Leave two words of pad so dp-4
                        ; will be zero and so $global$ will
 .WORD 0                ; still be double-word aligned
$global$                ; Contents of dp
```

Figure 6-2: Example Implementation of Start-up Code (assembly section)

```
__dld_sp
 .WORD 0
 .export __dld_sp, data

; 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
   .end
```

Figure 6-2: Example Implementation of Start-up Code (assembly section)

```
#include <sys/fcntl.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/mman.h>
#include <shl.h>

#define  T_MAP     (MAP_SHARED)
#define  T_PROT    (PROT_READ|PROT_EXECUTE)
#define  D_PROT    (PROT_READ|PROT_WRITE|PROT_EXECUTE)
#define  D_MAP     (MAP_PRIVATE)
#define  B_PROT    (PROT_READ|PROT_WRITE|PROT_EXECUTE)
#define  B_MAP     (MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED)
#define AOUTSIZE  sizeof( struct som_exec_auxhdr )

extern int _etext, _end, __dld_sp;
extern int __text_start, __data_start;
int __dld_loc;

__map_dld(progname)
   char* progname;
{
   static char MMAP_FAILED[] = "ERROR: mmap failed for dld";
   int fd, dev0_fd = -1;
   int tsize, dsize, bsize;
   struct header somheader;
   som_exec_auxhdr auxheader;
   char *taddr, *daddr, *baddr;
   int (*shl_init_ptr)();
   int result;
   extern void error();
   struct dld_parms parm;

   /* open dld.sl */
   if((fd = open("/usr/shlib/dld.sl", O_RDONLY)) == -1)
      error("ERROR couldn't open dld.sl", NULL, TRUE);

   if(read(fd, &somheader, sizeof (struct header)) \
      != sizeof (struct header))
      error("ERROR reading dld.sl", NULL, TRUE);
```

Figure 6-3: Example Implementation of Start-up Code (C section)

```
/* check magic number and system id */
   if(somheader.a_magic != 0x10E || somheader.system_id \
      != HP9000S800_ID)
      error("ERROR bad magic number/system id for dld.sl", \
              NULL, FALSE);

/* seek to aux_headers -- first one is exec aux header */
   lseek(fd, somheader.aux_header_location, 0L);
   if ((read(fd, &auxheader, AOUTSIZE) != AOUTSIZE
   || (auxheader.som_auxhdr.type != HPUX_AUX_ID))
      error("ERROR bad dld.sl exec aux header", NULL, FALSE);

   tsize = auxheader.exec_tsize;
   dsize = auxheader.exec_dsize;
   bsize = auxheader.exec_bsize;

   if ((taddr = mmap(0, tsize, T_PROT, T_MAP, fd, \
                  (off_t)auxheader.exec_tfile)) == -1)
      error(MMAP_FAILED, " (text)", TRUE);

/* map data */
   if ((daddr = mmap(0, dsize, D_PROT, D_MAP, fd, (off_t) \
                  auxheader.exec_dfile)) == -1)
      error(MMAP_FAILED, " (data)", TRUE);

/* now map bss */
   baddr = daddr + dsize;
   if (bsize != 0)
      if ( mmap( baddr, bsize, B_PROT, B_MAP, dev0_fd,
                  (off_t) 0) != baddr)
         error(MMAP_FAILED, " (bss)", TRUE);

/* now we can close the files */
   close(fd);

/* Put the parameters to _dld_main in a struct. The 1st parm will
   be (-1) to signify the dld_parms struct is used. The 2nd parm
   will be a ptr to that struct. The rest of the parms are used
   as before.   */
   parm.version = 0;
```

Figure 6-3: Example Implementation of Start-up Code (C section)

```
    parm.text_addr = taddr;
    parm.text_end = taddr+tsize;
    parm.prog_data_addr = &__data_start;

/* Now set up the call to dld's dld_main routine */
    shl_init_ptr = (int (*)()) ((int)taddr +
                auxheader.exec_entry-auxheader.exec_tmem);

/* need the equivalent of an import stub here to set up LTP */
    __dld_loc= daddr;

/* make the call */
    result = (*shl_init_ptr)(PARMS_STRUCT_USED, &parm, daddr, \
                baddr, baddr+bsize, &__text_start, &_etext, \
                progname, &_end, __dld_sp);
}

static void error(str, str2, errno_flg)
    char *str;
    char *str2;
    int errno_flg;
{
    char buf[10];

    write(2, "crt0: ", 6);
    write(2, str, strlen(str));

    if (str2 != NULL)
       write(2, str2, strlen(str2));

    if (errno_flg) {
       itoa(errno, buf);
       write(2, " errno:", 7);
       write(2, buf, strlen(buf));
    }

    write(2, "\n", 1);

    __exit(errno);
}
```

Figure 6-3: Example Implementation of Start-up Code (C section)

```
static int strlen(str)
   char *str;
{
   char *s = str;
   int count = 0;

   if (str == 0)
      return (0);

   while (*s++ != '\0')
      count++;

   return (count);
}


/* fills in the first 10 chars with ascii equiv of 'num',
   padded by zeros
*/
static itoa(num,buf)
   int num;
   char *buf;
{
   int cnt;

   buf[9] = '\0';

   for (cnt=8;cnt>=0;cnt--) {
      buf[cnt] = (num % 10) + '0';
      num /= 10;
   }
}
```

Figure 6-3: Example Implementation of Start-up Code (C section)

Linking and the Dynamic Loader

Applications need to be portable across a variety of systems, yet these systems are allowed to vary in implementation. This requires the implementation-specific areas to be provided with each system, yet readily accessible to a conforming application. This is possible through the use of dynamic libraries and incomplete executables.

The dynamic loader, /usr/shlib/dld.sl, is invoked by the start-up code (see Figure 6-2) when an incomplete executable is run. It binds the incomplete executable to the dynamic libraries according to options specified at link-time.

Program Linking

When relocatable objects are combined with libraries to form an executable, the linker must resolve all unsatisfied symbols. To assure that all linkers will perform as expected, the following set of rules for symbol resolution are specified:

- Unsatisfied symbols will be resolved to a local scope symbol in preference to a universal scope symbol in another module.
- Only symbols of matching type, name and name qualifier (if any) can be resolved to each other.
- Libraries are searched in the order they are specified on the command line. (This requires that a library containing an external reference must precede the library with the definition.)
- The first matching universal scope symbol encountered is used to resolve an unsatisfied symbol.

These rules apply to both archive and dynamic libraries.

When the linker resolves a symbol to a dynamic library routine, the linker binds all references to entries in a linkage table. This linkage table serves as a jump table once the dynamic loader maps the libraries and fills in the entries. Data items in the dynamic libraries are copied into the program executable so that the data references can be resolved statically.

The PRO standards do not currently specify the linker options used to support the functionality described in this section, but all conforming linkers must support the optional functionality. Specific linker options may be adopted in a future draft.

The Dynamic Loader

The dynamic loader is itself a dynamic library, although it does not define any symbols for use by a conforming application. This allows the application start-up code to use the SOM header and exec auxiliary header when mapping the dynamic loader into shared memory and calculating the address of the main entry point.

Once invoked, the dynamic loader attaches to the process all of the dynamic libraries that were linked with the application. The dynamic loader also resolves all symbolic references between the application and the dynamic libraries. Both of these functions may be controlled by options specified at link-time.

Library Location

When an incomplete executable is linked, a fully-qualified list of all of the libraries used is generated. The linker records this list in the executable (see "Dynamic Library List" on page 5-42). The linker may specify that these fully-qualified libraries are used at load time, or to use a library found using a search path.

The search path is a list of directories, with each entry separated by a colon. The dynamic loader searches for the libraries using the basenames found in the dynamic library list. The search path may be specified at link time and recorded in the *dl_header*, or contained in the environment variable SHLIB_PATH. The linker should allow for both methods to be used separately or together and in either order.

The search path itself uses the syntax of the shell PATH environment variable, a colon separated list of directories. The full library path name stored in the library list is referred to as the default library path and is represented in the library search path as a null entry (::).

For example, if a program is linked with the library /usr/shlib/libc.sl and the library search path is /users/campbelr/lib::old/lib, the dynamic loader will search for the libraries /users/campbelr/lib/libc.sl, /usr/shlib/libc.sl, and then ./old/lib/libc.sl.

If the dynamic loader cannot find a required library in any of the directories specified in the library search list, it will search for the library using the default library path.

Note The PRO ABI does not make any special provisions regarding security issues. The creator of setuid or setgid programs must ensure that users cannot substitute their own library on a search path and gain unintended privileges.

Binding

With all of the libraries identified, the dynamic loader must bind the symbolic references between the application and dynamic libraries. While the rules for binding are specified above, there are additional linker options to specify when binding occurs.

References to variables and other absolute address references are bound on the first resolution of a function call that could potentially reference the object. The function calls may be bound when the application is executed (BIND_IMMEDIATE) or when first called (BIND_DEFERRED). The binding method to be used is specified for each library in the dynamic library list by the linker.

The linker must provide options to allow either binding method to be specified. Conforming systems may substitute BIND_IMMEDIATE behavior for BIND_DEFERRED, but must provide BIND_IMMEDIATE behavior when it is requested.

When a dynamic library contains its own dynamic library list, this list is searched before the next library specified in the library list of the application.

Version Control

Since code from a dynamic library is mapped in at run time from a separate dynamic library file, modifications to a dynamic 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 address this issue.

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 Dynamic Library Version Auxiliary Header (See "String Areas" on page 5-25.) 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 dynamic 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.

Dynamic Library Initializers

A dynamic library can have an initialization routine, known as an *initializer*, that is called when the dynamic library is loaded or unloaded. Typically, an initializer is used to initialize a dynamic library's data when the library is loaded. The initializer is called for libraries that are loaded implicitly (at program start-up) or explicitly (via shl_load()). The initializer is also called when a library is explicitly unloaded.

When calling initializers for implicitly loaded libraries, the dynamic loader waits until all libraries have been loaded before calling the initializers. Initializers are called in depth-first order, the reverse order in which the libraries are searched for symbols. All initializers are called before the main program begins execution.

When calling the initializer for an explicitly loaded library, the dynamic loader waits until any dependency libraries are loaded before calling the initializers. As with implicitly loaded libraries, initializers are called in depth-first order.

Note that initializers can be disabled for explicitly loaded libraries via the BIND_NOSTART flag to shl_load.

Declaring the Initializer

Conforming linkers must specify an option to declare the name of the initializer. The dynamic library must reference the initializer, causing the linker to define an entry in the dynamic library import list. The linker shall set the INIT_DEFINED bit of the *flags* field of the dynamic library header, and set the *initializer* field to point to the appropriate import list entry (see "DL Header Fields" on page 5-32).

The actual definition of the initializer can appear in the dynamic library or in the main program. For instance, suppose *init_foo* is defined in libfoo.sl. To ensure that *init_foo* is registered as the initializer, you could include the following line in the library's source:

```
void (*init_foo_ptr)() = init_foo;
```

If, on the other hand, init_foo is defined outside the library (say, in the main program), you would need to declare *init_foo* as an external symbol:

```
extern void init_foo();
void (*init_foo_ptr)() = init_foo;
```

Initializer Syntax

The initializer routines shall have the following structure:

void initializer(shl_t handle, int loading);

With the arguments defined as follows:

initializer

The name of the initializer as specified to the linker.

handle

The initializer is called with this parameter set to the handle of the dynamic library for which it was invoked.

loading

The initializer is called with this parameter set to (-1) when the dynamic library is loaded and 0 when the library is unloaded.

Multiple Initializers

If multiple initializers are defined, they will be executed during the loading of a dynamic library in the same order specified on the command line when the library was built. Upon an explicit unload of a dynamic library, the initializers will be executed in the reverse of the order used during the loading of the library.

7

Standard Libraries

The standard system dynamic libraries reside in the /usr/shlib directory on all conformant systems. These libraries provide access to the functionality listed in the PRO API. While the interface to these libraries is defined by the PRO ABI, and the functionality provided is defined by the PRO API, the specific implementation of the libraries on each system is left unspecified. No conforming application may depend on functionality in any of the system libraries that is not included in the PRO standards.

Library Conventions

Library Names

The standard location for the system dynamic libraries is in the /usr/shlib directory. The location of the libraries must be known to a dynamically-linked executable. All conforming applications must be linked with the /usr/shlib libraries, or include the /usr/shlib directory in the library search path. See "Linking and the Dynamic Loader" on page 6-15.

Except when stated otherwise, system libraries must be provided as dynamic libraries. These are used with conforming applications using both the incomplete executable and relocatable object formats. Dynamic libraries shall use the ".sl" suffix.

Conforming systems may optionally provide archive libraries. These libraries are statically linked with relocatable objects. The resulting executable may not conform to the PRO ABI and may not be portable across different implementations of PRO compliant systems. Archive libraries will use the ".a" suffix.

The standard libraries that are provided by conforming systems are listed in Table 7-1.

Table 7-1: Standard System Libraries

Library	Description
libc	Standard system library
libM.a	Math function library[1]
libdld.sl	Dynamic library support library[2]
dld.sl	Dynamic loader[2]

1. The libM library must be used as an archive library.
2. These libraries are only provided as dynamic libraries.

Synonyms

Each entry point and global external data object is referenced by a symbol. These symbols are listed in the tables that follow for each library. The modules in the dynamic libraries provided on conforming systems must not make any reference to symbols that are within the ANSI C name space.

Many symbols are given synonyms that are outside the ANSI C name space. These synonyms may be used by the system libraries and must not be redefined by conforming applications.

The libc Library

Most of the system functionality is contained within the libc library. Entry points with synonyms are listed in Table 7-2, and without synonyms in Table 7-3.

The libc library also contains several data symbols that may be used by conforming applications. These global, exported data symbols appear in Table 7-4.

While no system libraries require or support the use of the long double data type, it is included in the ANSI C language. The routines listed in "Quad-precision Emulation Routines" on page 7-9 are used to support the operations (basic math, casting) specified as a part of the ANSI C language.

Table 7-2: Entry Points in libc with Synonyms

_assert	_filbuf	_flsbuf	abort
abs	access	alarm	asctime
atexit	atof	atoi	atol
bsearch	calloc	catclose	catgets
catopen	cfgetispeed	cfgetospeed	cfsetispeed
cfsetospeed	chdir	chmod	chown
chroot	clearenv	clearerr	clock
close	closedir	confstr	creat
ctermid	ctime	cuserid	difftime
div	drand48	dup	dup2
erand48	execl	execle	execlp
execv	execve	execvp	exit[1]
fchmod	fchown	fclose	fcntl
fdopen	feof	ferror	fflush

Table 7-2: Entry Points in libc with Synonyms

fgetc	fgetpos	fgets	fgetwc
fgetws	fileno	fnmatch	fopen
fork	fpathconf	fprintf	fprintmsg
fputc	fputs	fputwc	fputws
fread	freopen	frexp	fscanf
fseek	fsetpos	fstat	fsync
ftell	ftruncate	ftw	fwrite
getc	getchar	getclock	getcwd
getegid	getenv	geteuid	getgid
getgrgid	getgrnam	getgroups	getlogin
getopt	getpass	getpgrp	getpid
getppid	getpwnam	getpwuid	gets
gettimer	getuid	getw	getwc
getwchar	glob	globfree	gmtime
hcreate	hdestroy	hsearch	iconv
iconv_close	iconv_open	ioctl	isalnum
isalpha	isascii	isatty	iscntrl
isdigit	isgraph	islower	isprint
ispunct	isspace	isupper	iswalnum
iswalpha	iswcntrl	iswctype	iswdigit
iswgraph	iswlower	iswprint	iswpunct
iswspace	iswupper	iswxdigit	isxdigit

Table 7-2: Entry Points in libc with Synonyms

jrand48	kill	labs	lcong48
ldexp	ldiv	lfind	link
localeconv	localtime	longjmp	lrand48
lsearch	lseek	lstat	madvise
mblen	mbstowcs	mbtowc	memccpy
memchr	memcmp	memcpy	memmove
memset	mkdir	mkfifo	mktime
mktimer	mmap	modf	mprotect
mrand48	msem_init	msem_lock	msem_remove
msem_unlock	msgctl	msgget	msgrcv
msgsnd	msync	munmap	nice
nl_langinfo	nrand48	open	opendir
pathconf	pause	pclose	perror
pipe	poll	popen	printf
ptrace	ptsname	putc	putchar
putenv	puts	putw	putwc
putwchar	qsort	raise	rand
read	readdir	readlink	regcomp
regerror	regexec	regfree	reltimer
remove	rename	rewind	rewinddir
rmdir	rmtimer	sbrk	scanf
seed48	seekdir	semctl	semget

Table 7-2: Entry Points in libc with Synonyms

semop	setbuf	setclock	setgid
setgroups	setjmp	setlocale	setpgid
setsid	setuid	setvbuf	shmat
shmctl	shmdt	shmget	sigaction
sigaddset	sigdelset	sigemptyset	sigfillset
sigismember	siglongjmp	signal	sigpending
sigprocmask	sigsetjmp	sigsuspend	sleep
sprintf	srand	srand48	sscanf
stat	strcat	strchr	strcmp
strcoll	strcpy	strcspn	strerror
strfmon	strftime	strlen	strncat
strncmp	strncpy	strpbrk	strptime
strrchr	strspn	strstr	strtod
strtok	strtol	strtoul	strxfrm
swab	symlink	sysconf	system
tcdrain	tcflow	tcflush	tcgetattr
tcgetpgrp	tcsendbreak	tcsetattr	tcsetpgrp
tdelete	telldir	tempnam	tfind
time	times	tmpfile	tmpnam
toascii	tolower[2]	toupper[2]	towlower
towupper	truncate	tsearch	ttyname
twalk	tzset	ulimit[2]	umask

Table 7-2: Entry Points in libc with Synonyms

uname	ungetc	ungetwc	unlink
utime	vfprintf	vprintf	vsprintf
wait	waitpid	wcscat	wcschr
wcscmp	wcscoll	wcscpy	wcscspn
wcsftime	wcslen	wcsncat	wcsncmp
wcsncpy	wcspbrk	wcsrchr	wcsspn
wcstod	wcstok	wcstol	wcstombs
wcstoul	wcswcs	wcswidth	wcsxfrm
wctomb	wctype	wcwidth	wordexp
wordfree	write		

1. The synonym for exit is ___exit due to the alternate entry point previously defined as _exit.
2. These symbols have synonyms beginning with "__".

Table 7-3: Entry Points in libc without Synonyms

_exit	free	malloc	realloc

Table 7-4: Global External Data Symbols in libc

int __nl_char_size	FILE _iob[][1]	int daylight[2]
char **environ[1]	int errno	char *optarg
int opterr[1]	int optind[1]	int optopt[1]
long timezone[2]	char *tzname[][2]	

1. These symbols have synonyms beginning with "_".
2. These symbols have synonyms beginning with "__".

Quad-precision Emulation Routines

The routines listed in below are used to emulate instructions that are not implemented in hardware on all conforming systems. These instructions are defined in the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual, second edition*, and must comply to IEEE-754.

All routines listed in this section are defined as entry points in libc. These functions do not have any defined synonyms.

long double _U_Qfabs(long double x);

Emulate the FABS,QUAD r,t instruction. The value from register r is passed as argument x, and the value for register t is returned.

long double _U_Qfadd(long double x, long double y);

Emulate the FADD,QUAD r1,r2,t instruction. The value from register r1 is passed as argument x, the value from register r2 is passed as argument y, and the value for register t is returned.

void _U_Qfcmp(long double x, long double y, unsigned int condition);

Emulate the FCMP,QUAD,cond r1,r2 instruction. The value from register r1 is passed as argument x, the value from register r2 is passed as argument y, and the comparison flags cond are passed, in bits 27- 31 of condition.

The values for condition are as follows:

EXCEPTION	0x01
UNORDERED	0x02
EQUAL	0x04
LESSTHAN	0x08
GREATERTHAN	0x16

No value is returned, the result of the comparison is indicated by the C-bit in the Floating-Point Status Register.

long double _U_Qfcnvff_sgl_to_quad(float x);

Emulate the FCNVFF,SGL,QUAD r,t instruction. The single precision value in register r is passed as x, converted to quad precision format, and returned.

long double _U_Qfcnvff_dbl_to_quad(double x);

Emulate the FCNVFF,DBL,QUAD r,t instruction. The double precision value in register r is passed as x, converted to quad precision format, and returned.

float _U_Qfcnvff_quad_to_sgl(long double x);

Emulate the FCNVFF,QUAD,SGL r,t instruction. The quad precision value in register r is passed as x, converted to single precision format, and returned.

double _U_Qfcnvff_quad_to_dbl(long double x);

Emulate the FCNVFF,QUAD,DBL r,t instruction. The quad precision value in register r is passed as x, converted to double precision format, and returned.

long double _U_Qfcnvfx_sgl_to_quad(float x);

Emulate the FCNVFX,SGL,QUAD r,t instruction. The single precision floating-point value in register r is passed as x, converted to quad binary, fixed-point format, and returned.

long double _U_Qfcnvfx_dbl_to_quad(double x);

Emulate the FCNVFX,DBL,QUAD r,t instruction. The double precision floating-point value in register r is passed as x, converted to quad binary, fixed-point format, and returned.

unsigned integer _U_Qfcnvfx_quad_to_sgl(long double x);

Emulate the FCNVFX,QUAD,SGL r,t instruction. The quad precision floating-point value in register r is passed as x, converted to integer format, and returned.

double _U_Qfcnvfx_quad_to_dbl(long double x);

Emulate the FCNVFX,QUAD,DBL r,t instruction. The quad precision floating-point value in register r is passed as x, converted to double binary fixed-point format, and returned.

long double _U_Qfcnvfx_quad_to_quad(long double x);

Emulate the FCNVFX,QUAD,QUAD r,t instruction. The quad precision floating-point value in register r is passed as x, converted to quad binary fixed-point format, and returned.

long double _U_Qfcnvfxt_sgl_to_quad(float x);

Emulate the FCNVFXT,SGL,QUAD r,t instruction. The single precision floating-point value in register r is passed as x and converted to quad binary, fixed-point format and returned. The return value is rounded towards zero regardless of currently specified rounding mode.

long double _U_Qfcnvfxt_dbl_to_quad(double x);

Emulate the FCNVFXT,DBL,QUAD r,t instruction. The double precision floating-point value in register r is passed as x, converted to quad binary fixed-point format, and returned. The return value is rounded towards zero regardless of the currently specified rounding mode.

unsigned integer _U_Qfcnvfxt_quad_to_sgl(long double x);

Emulate the FCNVFXT,QUAD,SGL r,t instruction. The quad precision floating-point value in register r is passed as x, converted to integer format, and returned. The return value is rounded towards zero regardless of the currently specified rounding mode.

double _U_Qfcnvfxt_quad_to_dbl(long double x);

Emulate the FCNVFXT,QUAD,DBL r,t instruction. The quad precision floating-point value in register r is passed as x, converted to double binary fixed-point format, and returned. The return value is rounded towards zero regardless of the currently specified rounding mode.

long double _U_Qfcnvfxt_quad_to_quad(long double x);

Emulate the FCNVFXT,QUAD,QUAD r,t instruction. The quad precision floating-point value in register r is passed as x, converted to quad binary fixed-point format, and returned. The return value is rounded towards zero regardless of the currently specified rounding mode.

long double _U_Qfcnvxf_sgl_to_quad(unsigned int x);

Emulate the FCNVXF,SGL,QUAD r,t instruction. The single binary fixed-point value in register r is passed as x, converted to quad precision floating-point format, and returned.

long double _U_Qfcnvxf_dbl_to_quad(double x);

Emulate the FCNVXF,DLB,QUAD r,t instruction. The double binary fixed-point value in register r is passed as x, converted to quad precision floating-point format, and returned.

float _U_Qfcnvxf_quad_to_sgl(long double x);

Emulate the FCNVXF,QUAD,SGL r,t instruction. The quad binary fixed-point value in register r is passed as x, converted to single precision floating-point format, and returned.

double _U_Qfcnvxf_quad_to_dbl(long double x);

Emulate the FCNVXF,QUAD,DBL r,t instruction. The quad binary fixed-point value in register r is passed as x, converted to double precision floating-point format, and returned.

long double _U_Qfcnvxf_quad_to_quad(long double x);

Emulate the FCNVXF,QUAD,QUAD r,t instruction. The quad binary fixed-point value in register r is passed as x, converted to quad precision floating-point format, and returned.

long double _U_Qfdiv(long double x, long double y);

Emulate the FDIV,QUAD r1,r2,t instruction. The value from register r1 is passed as argument x, the value from register r2 is passed as argument y, and the value for register t is returned.

long double _U_Qfmpy(long double x, long double y);

Emulate the FMPY,QUAD r1,r2,t instruction. The value from register r1 is passed as argument x, the value from register r2 is passed as argument y, and the value for register t is returned.

long double _U_Qfrnd(long double x);

Emulate the FRND,QUAD r,t instruction. The value from register r is passed as argument x and the value for register t is returned.

long double _U_Qfsqrt(long double x);

Emulate the FSQRT,QUAD r,t instruction. The value from register r is passed as argument x and the value for register t is returned.

long double _U_Qfsub(long double x, long double y);

Emulate the FSUB,QUAD r1,r2,t instruction. The value from register r1 is passed as argument x, the value from register r2 is passed as argument y, and the value for register t is returned.

Additional Quad-Precision Routines

The following routines do not emulate instructions, yet are useful when working with quad precision math.

long double _U_Qfmax(long double x, long double y);

Returns the larger of two quad precision floating-point values.

long double _U_Qfmin(long double x, long double y);

Returns the smaller of two quad precision floating-point values.

long double _U_Qfrem(long double x, long double y);

Returns the remainder from the division of x by y.

The libM.a Library

The libM.a library contains all of the ANSI C math functions. Table 7-5 lists the entry points found in libM, and denotes those that have synonyms. Table 7-5 lists the global, external data symbols found in libM.a and denotes those with synonyms.

Table 7-5: Entry Points in libM.a

acos	asin	atan	atan2	ceil
cos	cosh	erf[1]	erfc[1]	exp
fabs	floor	fmod	gamma[1]	hypot[1]
isnan[1]	j0[1]	j1[1]	jn[1]	lgamma[1]
log	log10	pow	sin	sinh
sqrt	tan	tanh	y0[1]	y1[1]
yn[1]				

1. These symbols have a synonym beginning with "_".

Table 7-6: Global External Data Symbols in libM.a

signgam[1]

1. This symbol has a synonym beginning with "_".

The libdld.sl Library

The library /usr/shlib/libdld.sl library contains routines that are used to access dynamic libraries directly. This library is only provided as a dynamic library. No synonyms are provided for the libdld entry points, which are listed in Table 7-7.

Table 7-7: Entry Points in libdld

shl_definesym	shl_findsym	shl_get	shl_gethandle
shl_getsymbols	shl_load	shl_unload	

8 Commands & Execution Environment

While the PRO ABI does not attempt to describe a specific implementation of a system, certain details must be defined to support portable applications. This chapter specifies important files and directories, additional information regarding system commands and utilities, and how applications can perform initialization functions on both a system and per-user basis.

The commands listed in this chapter provide the functionality indicated by the PRO API. This document only describes required functionality not included in API standards.

Required Users and Groups

Conforming systems are required to define the users listed in Table 8-1 and groups defined in Table 8-2.

Table 8-1: Required User Names

bin	lp	root[1]

1. The user root is required to have user id (uid) of 0.

Table 8-2: Required Group Names

bin	daemon	mail
root[1]	sys	

1. The group root is required to have group id (gid) of 0.

Required Files and Directories

While the PRO ABI attempts to be independent of system implementation, certain files must be specified to support the portability of binaries.

/dev/console rw--w--w- root sys

Writes to /dev/console are printed to the system console.

/dev/null rw-rw-rw- bin bin

Also known as the "bit-bucket", /dev/null is a special file that always discards any data written to it, and always returns 0 bytes to any read.

/dev/ptym/clone rw-rw-rw- bin bin

Attempts to open /dev/ptym/clone shall return an open file descriptor of a free master pty device. If there are no free devices, the open shall return a value of (-1) and set the variable errno to EBUSY. The name of the slave device corresponding to the opened master device can be found through a ptsname() request.

/dev/tty rw-rw-rw- bin bin

The file /dev/tty is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs or shell sequences that need to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

/dev/ptym/pty[a-ce-z][0-9a-f] rw-rw-rw- bin bin

/dev/pytm[a-ce-z][0-9][0-9] rw-rw-rw- bin bin

/dev/pty[pqr][0-9a-f] rw-rw-rw- bin bin

These files are the master pseudo terminals. For more information, see "pty - Pseudo Terminal Drivers" on page 9-2.

`/dev/pty/tty[a-ce-z][0-9][0-9]`	`rw-rw-rw-`	`bin`	`bin`
`/dev/pty/tty[a-ce-z][0-9a-f]`	`rw-rw-rw-`	`bin`	`bin`
`/dev/tty[pqr][0-9a-f]`	`rw-rw-rw-`	`bin`	`bin`

These files are the slave pseudo terminals. For more information, see "pty - Pseudo Terminal Drivers" on page 9-2.

`/sbin/init.d/` `rwxrwxr-x` `bin` `bin`

This directory contains any scripts to be executed when the system transitions to a new run-level.

`/sbin/rc#.d` `rwxrwxr-x` `bin` `bin`

These directories (where # substitutes for the run-level) contains links to the scripts contained in /sbin/init.d and controls their execution. See "User-shell Initialization" on page 8-13.

`/tmp` `rwxrwxrwx` `bin` `bin`

A directory for the storage of temporary files. It is recommended that applications use the /var/tmp or /var/opt/<application> directories.

`/usr/lib` `rwxrwxr-x` `bin` `bin`

This is the default location for any system relocatable libraries.

`/usr/shlib` `rwxrwxrwx` `bin` `bin`

This directory is the default location for all system dynamic libraries. It is also the required location of the dynamic loader, dld.sl. For historical reasons, systems may desire to create a symbolic link from /lib to /usr/shlib.

`/usr/shlib/dld.sl` `r-xr-xr-x` `bin` `bin`

The dynamic loader. See "The Dynamic Loader" on page 6-16.

`/var/opt` `rwxrwxr-x` `bin` `bin`

Applications are allowed to place a subdirectory in this directory using the same name used in the /opt directory. This subdirectory could then be used to store application specific data, log, or temporary files.

`/var/tmp` **`rwxrwxrwx bin bin`**

The /var/tmp directory is for storage of application-generated temporary files. It differs from /tmp in that its contents are to be preserved across system reboots.

Commands

The commands listed in Table 8-3 are located in the directory /usr/bin (unless noted otherwise) on conforming system and are available for use in a conforming applications shell scripts.

Table 8-3: Commands and Utilities on Conforming Systems

admin	alias[1]	ar	asa	at
awk	basename	batch	bc	bg[1]
c89	cal	calendar	cancel	cat
cc	cd[1]	cflow	chgrp	chmod
chown	cksum	cmp	col	comm
command[1]	compress	cp	cpio	cpp
crontab	csplit	ctags	cu	cut
cxref	date	dd	delta	df
diff	dircmp	dirname	du	echo
ed	egrep	env	ex	expand
expr	false	fc[1]	fg[1]	fgrep
file	find	fold	fort77	gencat
get	getconf	getopts[1]	grep	hash
head	iconv	id	jobs[1]	join
kill	lex	line	lint	ln
locale	localedef	logger	logname	lp
lpstat	ls	m4	mail	mailx
make	man	mesg	mkdir	mkfifo
mknod	more	mv	newgrp	nice

Table 8-3: Commands and Utilities on Conforming Systems

nl	nm	nohup	od	pack
paste	patch	pathchk	pax	pcat
pg	pr	printf	prs	ps
pwd	read[1]	renice	rm	rmdel
rmdir	sact	sccs	sed	sh
sleep	sort	spell	split	strings
strip	stty	sum	tabs	tail
talk	tar	tee	test[1]	time
touch	tput	tr	true	tsort
tty	type	ulimit	umask[1]	unalias[1]
uname	uncompress	unexpand	unget	uniq
unpack	uucp	uudecode	uuencode	uulog
uuname	uupick	uustat	uuto	uux
val	vi	wait[1]	wc	what
who	write	xargs	yacc	zcat

1. These commands are shell built-ins, and are not to be used with a full pathname.

Options for c89

The functionality specified in the PRO ABI requires that the c89 compiler supports additional options not specified in the PRO API standards. These options are listed in Table 8-4.

Table 8-4: Nonstandard c89 Options

Option	Description
-b	Create a dynamic library rather than a normal executable file. Objects processed with this option must contain position-independent code.
-n	This flag indicates that the resulting file shall be a sharable, executable SOM. (SHARE_MAGIC)
-N	Create a non-shareable executable (EXEC_MAGIC)
-q	This flag indicates that the resulting file shall be a sharable, demand-loadable, executable SOM. (DEMAND_MAGIC)
-z	Do not bind anything to address zero. This option allows for the run time detection of null pointers.
-Z	Allow dereferencing of null pointers.
-Wl,arg1[,arg2...]	This flag indicates that arguments should be passed to the linker. These arguments are listed below.
-a search	Specify whether dynamic or archive libraries are searched for the -l option. The value of search should be either archive (relocatable) or shared (dynamic). This option can appear more than once, interspersed among -l options, to control the searching for each library.
-l: library	Search the library specified. Similar to the compiler option -l, except the current state of the -a option is not important. The library name must contain the prefix lib and end with a suffix of .a or .sl.

Table 8-4: Nonstandard c89 Options

Option	Description
-u symbol	Enter symbol as an undefined symbol in the symbol table. The resulting unresolved reference is useful for linking a program solely from the object files in a library. this option may be specified several times, each with a single symbol
-B bind	Select the run-time binding behavior when using dynamic libraries. Options for bind are immediate or deferred. See "bind" on page 5-43.+b path_listSpecify a colon-separated list of directories to be searched at program run-time to locate the dynamic libraries that were specified with the -l or -W,-l: options at link time. An argument of a single colon (:) indicates that the linker must build the list using all directories specified by the -L option and the LPATH environment variable (see +s option).
+s	Indicates that the dynamic loader (dld.sl) can use the environment variable SHLIB_PATH to locate the dynamic libraries that were specified with the -l or -W,-l: options. SHLIB_PATH should be set to a colon-separated list of directories. If both +s and +b are used, their relative order on the command line indicates which path list will be searched first. See "flags" on page 5-34.

User and System Initialization

Conforming applications may require the ability to configure their execution environment. This may include the ability to start daemons, add a directory to the search list of the $PATH environment variable, or cleaning temporary files during a reboot. This section defines how an application can control tasks to be performed when the system or a new user session is started.

System Initialization Shell Scripts

Applications may require certain actions be performed when the system state changes. The system state is defined in terms of run-levels, with different system functionality available at each level. An application may specify a start script to be run when the system enters a new state or a kill script that is executed when the system returns to a lower run level.

Start and Kill Scripts

Start and kill scripts reside in the /sbin/init.d directory. In actual usage, the start and kill scripts may in fact be a single script, with the behavior controlled by the arguments received. These scripts are not directly executed, but are controlled through symbolic links.

These symbolic links are contained in sequencer directories, /sbin/rc#.d (where # is a run-level). When entering a higher level, all scripts in the sequencer directory associated with that run level that have names beginning with the letter "S" are invoked with the argument "start". When the system transitions to a lower run-level, all scripts in the sequencer directory associated with that run level that have names beginning with the letter "K" are invoked with the argument "stop". This will be explained further in the following section.

Naming Convention

The start and kill scripts used by conforming applications must begin with the underscore character "_". Conforming systems are not allowed to use the underscore character as the first letter of any system scripts.

The naming convention of the symbolic links in the sequencer directory is more complex. It is described using the following example:

```
           1  2 3        4
/sbin/rc3.d/S225_drive_server
```

Where the components of the pathname are interpreted as shown below:

1. This is the run-level for which the scripts contained in the /sbin/rc3.d directory are executed. In this example, start scripts will be executed upon entering run-level 3 from run-level 2. Kill scripts in the rc3.d directory will be executed upon entering run-level 3 from run-level 4.
2. The first character of a sequencer link name determines whether the script is executed as a start script (first character is "S") or as a kill script (first character is "K").
3. A three digit number is used for sequencing scripts within the sequencer directory. Scripts are executed by type (start or kill) in alphabetical order.
4. Following the sequence number is the name of the script. This name must match the script to which the sequencer is linked. In the above example, the link points to /sbin/init.d/_drive_server.

It is important to note that any kill script sequencer associated with the example start script /sbin/rc3.d/S225_drive_server will be located in the directory /sbin/rc2.d. The sequencer number and the script name may be different or stay the same.

Arguments to Scripts

Start and kill scripts should be able to recognize the following four arguments (where applicable) and take the action indicated:

- start. The "start" argument is passed to scripts whose names start with "S". Upon receiving the "start" argument, the script should perform its start actions.

- **stop.** The "stop" argument is passed to scripts whose names start with "K". Upon receiving the "stop" argument, the script should perform its stop actions.
- **start_msg.** The "start_msg" argument is passed to scripts whose names start with "S" to instruct the script to report back a short message indicating what the "start" action will do.
- **stop_msg.** The "stop_msg" argument is passed to scripts whose names start with "K" to instruct the script to report back a short message indicating what the "stop" action will do.

Script Output

All start and kill scripts are required to write all status messages to stdout, and all error messages to stderr. Scripts are not allowed to write directly to the system console, or to start daemons that immediately write to the console.

Exit Values

Start and kill scripts must use the convention for exit values listed in Table 8-5.

Table 8-5: Exit Values for Start and Kill Scripts

Exit Value	Definition
0	No errors occurred during script execution.
1	Script failed due to an error condition.
2	Script functions not executed due to some overriding condition.

Run Levels

As discussed previously, the various run levels denote differing states of the system. Not all run levels are defined by the PRO ABI. The run-levels at which a conforming application may use a start or kill script are listed in Table 8-6.

Table 8-6: Run Level Definitions

Run Level	Definition
1	Run level 1 is defined by the PRO ABI only as the level prior to run level 2. Conforming applications may only place kill scripts at this level.
2	This is the multi-user state run level. This is the lowest level that conforming applications may use for start scripts.
3	This is the run level for remote file access. Applications may expect that any NFS file systems are imported or exported at this level.

User-shell Initialization

Applications may require that certain initialization functions be performed when a user logs onto a system. For example, the environment variables $PATH and $SHLIB_PATH may need to have entries added for application specific executables and shared libraries. This ability is provided by the /etc/opt/profile.d directory.

If an application needs to modify the shells profile, it may include a shell script in its directory (/opt/<application>) and include a symbolic link in /etc/opt/profile.d. The link itself will share the name of the application directory (/opt/<application>.)

When a user logs onto a conforming system using sh, the system is required to process all files indicated by the links located in the /etc/opt/profile.d directory.

While applications are not allowed to use, and systems are not required to provide csh, an application may provide a script to provide initialization for csh users. This script will also exist in the applications directory (/opt/<application>) with a symbolic link to it located in /etc/opt/csh.login.d sharing the name of the application directory.

9

Terminal and Windowing Interfaces

This chapter is intended to provide support for only that functionality found on a base level system as defined by the PRO API. Full support for functionality specified in the options of the PRO API will be provided in supplements to this document.

pty - Pseudo Terminal Drivers

The pty driver provides support for a device-pair termed a pseudo terminal. A pseudo terminal is a pair of character devices, a master device and a slave device. The slave device provides to application processes an interface as is described in the "General Terminal Interface" chapter found in the *System Interface Definitions, Issue 4* of the X/Open CAE Specification. The slave device does not have a hardware device behind it. Instead, it has another process manipulating it through the master half of the pseudo terminal. Thus anything written on the master device is given to the slave device as input, and anything written on the slave device is presented as input on the master device. This is illustrated in Figure 9-1.

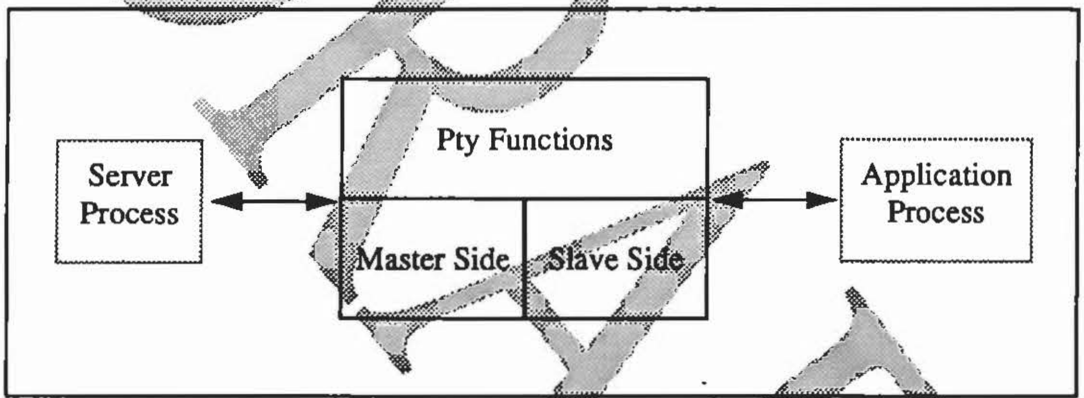

Figure 9-1: Pseudo Terminal Operation

Open and Close Processing

The slave side of the pty interprets opening or closing the master side as a modem connection or disconnection on a real terminal. Only one open to the master side of a pty is permitted. An attempt to open an already open master side returns (-1) and sets the external variable *errno* to EBUSY. An attempt to open the master side of a pty that has a slave with an open file descriptor returns (-1) and sets *errno* to EBUSY. The potential problem of ptys being found busy at open() calls can be avoided by using the clone open functionality discussed in the next section.

An attempt to open a non-existent pty returns (-1) and sets *errno* to ENXIO. If O_NONBLOCK is not specified, opens on the slave side hang until the master side is opened. An open() of a slave pty with a mode containing O_NONBLOCK when the master side of that pty is not open, shall return (-1) and set *errno* to EAGAIN. Any ioctl() or write() request made on the slave side of a pty after the master side is closed returns (-1) and sets the external variable *errno* to EIO. A read() request made on the slave side of a pty after the master side is closed returns 0 bytes. Closing the master side of a pty sends a SIGHUP hang-up signal to the calling process and flushes pending input and output.

Master/Slave Pairing

The master and slave pty device files that may be directly accessed on a conforming system are listed in "Required Files and Directories" on page 8-3. These special files are paired with the "tty" present in the file name of the slave being replaced with "pty" in the file name of the mater.

Both slave and master device file in a pty pair are required to have identical minor numbers.

Clone Open

In typical pty usage, there is no preference among pty pairs. Thus, it is useful to be able to issue a single open() that internally opens any available pty. An open on the clone device, /dev/ptym/clone, returns an open file descriptor of a free master pty device. If there are no free devices, the open returns (-1) and sets *errno* to EBUSY. The name of the slave device corresponding to the opened master device can be found through a ptsname() request.

Termios Special Character Initialization

The *c_cc* array in the *termios* structure (see "<termios.h>" on page B-51) defines special characters with defined functions. The initial values that a conforming system must provide as a default are listed below in Table 9-1.

Table 9-1: Initial Values for Termios Special Character Array

Character	c_cc[ ] Index	Default Value
EOF	VEOF	Control-D
EOL	VEOL	NUL
ERASE	VERASE	#
INTR	VINTR	DEL
KILL	VKILL	@
MIN	VMIN	NUL
QUIT	VQUIT	Control-|
START	VSTART	Control-Q
STOP	VSTOP	Control-S
SUSP	VSUSP	disabled
SWTCH	VSWTCH	NUL
TIME	VTIME	Control-D

The X Window System

The PRO API currently specifies that X Window System™, as is specified in the MIT X Consortium Standard: *X Window System Version 11, Release 5*. Conforming system that desire to support this option shall support the functionality specified in the *X Window System Protocol, Version 11 Specification*.

Full binary support for this option on PA-RISC will be defined in a supplement to the PRO ABI.

10

Relocatable Objects

While dynamic linking provides portability by allowing applications to load system-specific code at run time, there is an associated performance cost. Applications that wish to avoid the requirements of incomplete executables may do so by using relocatable objects.

Relocatable objects require the SOM Header, Exec Header, Text and Data areas as specified for incomplete executables. Relocatable objects additionally require that the Space Dictionary, Subspace Dictionary, Symbol Table, Relocation Information, Space and Symbol Strings Tables exist with the proper entries included in the SOM Header. It may optionally include compiler records and additional auxiliary headers.

Relocatable objects may be statically bound with archive libraries supplied by the target system to produce complete executables. These complete executables do not conform to the PRO ABI and are not guaranteed to be portable between systems.

This chapter contains the additional information required to support relocatable objects.

Dynamic Library Version Auxiliary Header

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

If a relocatable object does not contain a dynamic library version auxiliary header, the linker shall use a value of 0 for the object version.

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

Figure 5-1: Dynamic Library Version Auxiliary Header Definition

Dynamic Library Version Auxiliary Header Fields

header_id

This field contains the auxiliary header identifier for the dynamic library version header. See "Auxiliary Header Fields" on page 5-17.

version

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

Standard Spaces and Subspaces

The concept of a space is discussed in "Virtual Address Space" on page 3-12. Spaces are fundamental to the PA-RISC architecture. Subspaces are logical subdivision of a space used for grouping code and data.

Spaces and subspaces are ordered by a conforming linker using the *sort_key*. Linkers shall places spaces with lower *sort_key* values in front of spaces with higher *sort_key* values. Subspaces within a space are also sorted by ascending *sort_key* values.

The standard spaces, subspaces and sort keys are listed in Table 10-1.

Note

Conforming programs that create subspaces other than those listed in Table 10-1 should not use sort keys less than 8 or larger than 56 in either $TEXT$ or $PRIVATE$ spaces.

Table 10-1: Standard Spaces, Subspaces and Sort Keys

Space	Subspace	Sort Key	Use
$TEXT$		8	
	$SHLIB_INFO$	0	Dynamic library information
	$MILLICODE$	8	Millicode routines
	LIT	16	Literals
	$CODE$	24	Normal code
	$UNWIND_START$	56	Stack unwind table
	$UNWIND_END$	72	Stub stack unwind table
	$RECOVER_START$	73	Reserved
	$RECOVER_END$	88	Reserved
$PRIVATE$		16	
	$SHLIB_DATA$	16	Shared library data
	$DATA$	16	Global arrays and structures
	$SHORTDATA$	24	Global scalar variables
	$PLT	38[1]	Procedure linkage table
	DLT	39[2]	Data linkage table
	$GLOBAL$	40	Global variable base address
	$SHORTBSS$	80	Uninitialized data and common
	BSS	82	Uninitialized data and common

1. A sort key of 7 is used for the PLT subspace in a dynamic library.
2. A sort key of 6 is used for the DLT subspace in a dynamic library.

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. Conforming applications are limited to one code and one data space, with each space limited to a single quadrant. 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 {
    int             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    reserved1 : 13;         /* reserved */
    unsigned int    sort_key : 8;           /* sort key for space        */
    unsigned int    reserved2 : 8;          /* reserved */
    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;       /* reserved */
    unsigned int    loader_fix_quantity;    /* reserved */
    int             init_pointer_index;     /* index into init pointer array */
    unsigned int    init_pointer_quantity;  /* number of data (init) pointers */
};
```

Figure 10-2: Space Dictionary Record Definition

Space Dictionary Fields

name

The field *name* 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* is a byte offset relative to the field *space_strings_location* in the SOM header. *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 all bits to zero in *name* indicates a null name pointer. *name* must have a value in the range 0 to 2^{31}-1.

is_loadable

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

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

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

reserved1

This field is reserved for future use. It must be initialized to 0.

sort_key

This field specifies a sort key which may be used by the linker for ordering spaces in the output file. The first occurrence of each space defines *sort_key* for that space.

The linker arranges the spaces in ascending order using *sort_key*.

reserved2

These bits are reserved and must be filled with 0's.

space_number

This field specifies the number assigned to this space. Conforming loaders must 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 - loader_fix_quantity

These fields are currently reserved. Conforming applications must set both *loader_fix_index* to (-1) and *loader_fix_quantity* to 0.

init_pointer_index

This field is a zero-relative array index into the initialization pointer array. All initialization pointers for a space will be contiguous records in the initialization pointer array. If there are no initialization pointers for a space, *init_pointer_index* must be set to the value (-1).

If *init_pointer_index* is greater than *init_array_total* in the SOM header, it is an error.

In a relocatable object, this field shall be initialized to (-1).

init_pointer_quantity

This field is the number of initialization pointers in the space. If *init_pointer_index* + *init_pointer_quantity* is greater than *init_array_total* in the SOM header, it is an error.

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;      /* reserved */
    unsigned int    undefined :1;             /* undefined */
    unsigned int    code_only :1;             /* must contain only code  */
    unsigned int    sort_key :8;              /* subspace sort key       */
    unsigned int    replicate_init :1;        /* reserved */
    unsigned int    continuation :1;          /* reserved */
    unsigned int    reserved :6;              /* reserved */
    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 :16;            /* reserved */
    unsigned int    alignment :16;            /* alignment required for the
                                                 subspace (largest alignment
                                                 requested for any item in
                                                 the subspace)           */
    unsigned int    name;                     /* index of subspace name */
    int             fixup_request_index;      /* index into fixup array */
    unsigned int    fixup_request_quantity;   /* number of fixup requests */
};
```

Figure 10-3: Subspace Dictionary Record Definition

Subspace Dictionary Fields

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:

offset = *space_index* * size of (space record)
+ *space_dictionary_location* (found in the SOM header)
+ address of the first byte of the SOM header.

If a *space_index* is greater than the field *space_quantity* in the SOM header record 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. The layout and interpretation of the *access_control_bits* field is shown in Figure 10-4 and Table 10-2.

Type (3 bits)	PL1 (2 bits)	PL2 (2 bits)

Figure 10-4: Layout of *access_control_bits*

In a conforming application, text pages should use a value of 0x2c, and data pages should use 0x1f. Loaders on conforming systems must conform to the restrictions listed in "Virtual Address Space" on page 3-12.

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. Conforming applications must use a value of 0.

Table 10-2: Interpreting Access Control Bits

Type value (in binary)	Allowed access types and GATEWAY promotion	Privilege check
000	Read only data page	read: PL ≤ PL1 write: Not allowed execute: Not allowed
001	Normal data page	read: PL ≤ PL1 write: PL ≤ PL2 execute: Not allowed
010	Normal code page	read: PL ≤ PL1 write: Not allowed execute: PL2 ≤ PL ≤ PL1
011	Dynamic code page	read: PL ≤ PL1 write: PL ≤ PL2 execute: PL2 ≤ PL ≤ PL1
100	Gateway to PL0[1]	read: Not allowed write: Not allowed execute: PL2 ≤ PL ≤ PL1
101	Gateway to PL1[1]	read: Not allowed write: Not allowed execute: PL2 ≤ PL ≤ PL1
110	Gateway to PL2[1]	read: Not allowed write: Not allowed execute: PL2 ≤ PL ≤ PL1
111	Gateway to PL3[1]	read: Not allowed write: Not allowed execute: PL2 ≤ PL ≤ PL1

1. Change of privilege level only occurs if the indicated new value is of higher privilege than the current privilege level, otherwise the target of the GATEWAY executes at the same privilege as the GATEWAY itself.

dup_common

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

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

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

quadrant

This is to specify which of the four possible quadrants, numbered 0-3, 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.

initially_frozen

If this flag is set to one then the subspace is to be locked in physical memory when the operating system is being booted. Conforming applications must set this flag to 0.

undefined

is_first

This bit is not defined by the PRO ABI.

code_only

If set specifies that this subspace contains only code (no literal data).

sort_key

This field contains the primary sort key by which the linker arranges subspaces in an output file. Subspaces are first placed in ascending order by the sort key, then 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 - continuation - reserved

These bits are reserved for future use. They must be set to 0.

file_loc_init_value

If *initialization_length* field is non-zero, this field contains a byte offset relative to the first byte of the SOM header. The field *file_loc_init_value* points to the data used to initialize a subspace.

If *initialization_length* is zero then this field contains a 32 bit quantity which is used as an initialization pattern for the entire subspace.

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.

subspace_start

This is a byte address of where the subspace is to start relative to the beginning of a space. 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

These bits are reserved and must be set to 0.

alignment

This field specifies what alignment is required for the subspace 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. *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 *name* to zero means that it is a null name pointer. See "String Areas" on page 5-25.

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. The *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. The *fixup_request_index* must have a value in the range (-2^{31}) to $2^{31}-1$.

fixup_request_quantity

The *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. The *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.

Compilation Unit Dictionary

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 {
    unsigned int        name;
    unsigned int        language_name;
    unsigned int        product_id;
    unsigned int        version_id;
    unsigned int        reserved;
    unsigned int        compile_time[2];
    unsigned int        source_time[2];
};
```

Figure 10-5: Definition of Compilation Unit Dictionary Record

Compilation Unit Directory Fields

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 the 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. They must be set to 0's in a conforming application.

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.

Symbol Dictionary

The 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.

Symbol records do not have to be sorted.

```
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;
    unsigned int    name;
    unsigned int    qualifier_name;
    unsigned int    symbol_info;
    unsigned int    symbol_value;
};
```

Figure 10-6: Symbol Dictionary Record Definition

Symbol Dictionary Record Fields

hidden

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

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

This field defines what type of information this symbol represents. A complete list of the defined symbol types is presented in Table 10-3, however only certain ones may be valid depending on the use (e.g. import/export, relocatable/executable, etc.).

Table 10-3: ***symbol_type*** **Definition**

#	Symbol	Description
0	ST_NULL	Invalid symbol record. The contents of the entire record is undefined (it is 5 words long).
1	ST_ABSOLUTE	Absolute constant.
2	ST_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.

Table 10-3: ***symbol_type*** **Definition**

#	Symbol	Description
3	ST_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	ST_PRI_PROG	Primary program entry point.
5	ST_SEC_PROG	Secondary Program entry point.
6	ST_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	ST_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 the BSS or $SHORTBSS$ subspace by the linker and the symbol's value becomes the virtual address of that storage. The linker will convert the symbol to the ST_DATA type. The loader shall initialize the area with 0's. For example, Fortran and Cobol uninitialized common data blocks, and the C construct "`int i`" would be storage requests with no initial value.
8	ST_STUB	This symbol marks an import (outbound) external call stub or a parameter relocation stub. Created by the linker, this type will not be found in relocatable objects.
9	ST_MODULE	This symbol is a source module name.
12	ST_MILLICODE	This symbol is the name of a millicode routine.
13	ST_PLABEL	This symbol defines an export stub for a procedure for which a procedure label has been generated.

symbol_scope

The scope of a symbol defines the valid range for an exported symbol, 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:

SS_UNSAT	Import request that has not been satisfied.
SS_EXTERNAL	Used with ST_STUB to indicate an import stub.
SS_LOCAL	This symbol is not exported outside the SOM. It may be used as the target for fixups, but the linker does not use this symbol for resolving symbol references.
SS_UNIVERSAL	This symbol is exported for use outside the SOM.

Table 10-4 shows the valid values of the scope field given the type of the symbol. Elements not marked by an "X" are invalid values for that type.

Table 10-4: Valid *symbol_scope* Values

TYPE \ SCOPE	SS_UNSAT	SS_EXTERNAL	SS_LOCAL	SS_UNIV
ST_PRI_PROG				X
ST_SEC_PROG				X
ST_ENTRY			X	X
ST_STUB		X	X	
ST_MODULE			X	X
ST_ABSOLUTE	X		X	X
ST_CODE	X		X	X
ST_MILLICODE	X		X	X
ST_DATA	X		X	X
ST_STORAGE	X			
ST_PLABEL			X	

check_level

This field is reserved. It shall be initialized to 0.

must_qualify

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

This field is reserved. It shall be initialized to 0.

memory_resident

This field is reserved. It shall be initialized to 0.

is_common

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.

dup_common

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

This field is not defined by the PRO ABI.

arg_reloc

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 ST_ENTRY, ST_PRI_PROG, and ST_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

The argument location values are defined in Table 10-5

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.

Table 10-5: Argument Relocation Values

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.

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.

symbol_Info

This field contains the index of the subspace containing this symbol when the symbol_value is of type CVA, DVA, or LEN as indicated in Table 10-6. The *symbol_info* field is not defined in absolute symbols or in an executable.

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. Table 10-6 shows the meaning of the symbol value for each valid combination of type and scope. Invalid combinations will be denoted as a blank cell in the matrix. A *symbol_value* of (-1) indicates that the symbol is a duplicate and may be ignored. The definitions for the mnemonics used are:

CVA - This stands for code virtual address and it is the byte offset within a space (when it is loaded in virtual memory). 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 during execution).
DVA - This stands for data virtual address and it is the byte offset within a space (when it is loaded in virtual memory).
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.

Table 10-6: Valid *symbol_value* Mnemonics

TYPE	SS_UNSAT	SS_EXTERNAL	SS_LOCAL	SS_UNIV
ST_PRI_PROG				CVA
ST_SEC_PROG				CVA
ST_ENTRY			CVA	CVA
ST_STUB		UNUSED	CVA	
ST_MODULE			UNUSED	UNUSED
ST_ABSOLUTE	UNUSED		CONST	CONST
ST_CODE	UNUSED		CVA	CVA
ST_MILLICODE	UNUSED		CVA	CVA
ST_DATA	UNUSED		DVA	DVA
ST_STORAGE	LEN			
ST_PLABEL			DVA	

Fixups

In newer 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. Certain fixup requests cause the linker to copy one or more bytes from the input subspace to the output subspace without change, while others direct the linker to relocate words or resolve external references. Still others direct the linker to insert zeroes in the output subspace or to leave areas uninitialized without copying any data from the input subspace, and others describe points in the code without contributing any new data to the output file.

Many fixup requests use a range of opcodes; refer to Table 10-9 for a listing of opcode values.

Table 10-7: Rounding Modes used in Fixups

Field Selector	Meaning
L'	Arithmetic shift right 11 bits
R'	Set bits 0-20 to 0
LD'	Add 0x800, arithmetic shift right 11 bits
RD'	Set bits 0-20 to 1
LR'	Round constant before evaluating expression, arithmetic shift right 11 bits
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 arithmetic shift right 11 bits
RS'	Sign extend from bit 21

Fixup Requests

The meaning of each fixup request is described below. The opcode ranges and parameters for each fixup are described in Table 10-9.

Table 10-8: 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.
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.
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. The word is assumed to be a pc-relative procedure call instruction for example, BL). The target procedure is identified by symbol S, and the parameter relocation bits are R.
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.

Table 10-8: 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 GR 27. Instructions other than LDIL and ADDIL may have a small constant in the displacement field of the instruction.
R_DLT_REL	Copy one instruction word with relocation. The word is assumed to be a GR 19-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 GR 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). Instructions other than LDIL and ADDIL may have a small constant in the displacement field of the instruction.
R_MILLI_REL	This fixup is not currently defined for use in PRO compliant applications.
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.

Table 10-8: Fixup Requests

R_BREAKPOINT	Copy one instruction word conditionally. The linker must always replace the word with a NOP instruction.
R_ENTRY	Define a procedure entry point. The stack unwind bits, U, and the frame size, F, may be recorded in a stack unwind descriptor.
R_ALT_ENTRY	Define an alternate procedure entry point.
R_EXIT	Define a procedure exit point.
R_BEGIN_TRY	Define the beginning of a try/recover region.
R_END_TRY	Define the end of a try/recover region. The offset R defines the distance in bytes 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.
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.

Table 10-8: Fixup Requests

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_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.

Table 10-8: Fixup Requests

<table>
<tr><td>R_COMP1[1]</td><td>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.</td></tr>
<tr><td>R_PUSH_PCON1</td><td>Push the (positive) constant V.</td></tr>
<tr><td>R_PUSH_DOT</td><td>Push the current virtual address.</td></tr>
<tr><td>R_MAX</td><td>Pop A and B, then push max(A, B).</td></tr>
<tr><td>R_MIN</td><td>Pop A and B, then push min(A, B).</td></tr>
<tr><td>R_ADD</td><td>Pop A and B, then push A + B.</td></tr>
<tr><td>R_SUB</td><td>Pop A and B, then push B - A.</td></tr>
<tr><td>R_MULT</td><td>Pop A and B, then push A * B.</td></tr>
<tr><td>R_DIV</td><td>Pop A and B, then push B / A.</td></tr>
<tr><td>R_MOD</td><td>Pop A and B, then push B % A.</td></tr>
<tr><td>R_AND</td><td>Pop A and B, then push A & B.</td></tr>
<tr><td>R_OR</td><td>Pop A and B, then push A | B.</td></tr>
<tr><td>R_XOR</td><td>Pop A and B, then push A XOR B.</td></tr>
<tr><td>R_NOT</td><td>Replace A with its complement.</td></tr>
<tr><td>R_LSHIFT</td><td>If C = 0, pop A and B, then push B << A. Otherwise, replace A with A << C.</td></tr>
<tr><td>R_ARITH_RSHIFT</td><td>If C = 0, pop A and B, then push B >> A. Otherwise, replace A with A >> C. The shifting is done with sign extension.</td></tr>
<tr><td>R_LOGIC_RSHIFT</td><td>If C = 0, pop A and B, then push B >> A. Otherwise, replace A with A >> C. The shifting is done with zero fill.</td></tr>
<tr><td>R_PUSH_NCON1</td><td>Push the (negative) constant V.</td></tr>
</table>

Table 10-8: Fixup Requests

R_COMP2[1]	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.
R_PUSH_NCON2	Push the (negative) constant V.
R_COMP3[1]	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_RESERVED	Fixups in this range are reserved for internal use by the compilers and linker.

1. The secondary opcodes for the fixup requests R_COMP1, R_COMP2 and R_COMP3 are listed in Table 10-10, on page 10-39.

Table 10-9 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.

Parameter relocation bits are encoded in the fixup requests in two ways, noted as rbits1 and rbits2 in Table 10-9. 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.

The second encoding is more complex; 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.

Table 10-9: Fixup Request Opcodes and Parameters

mnemonic	opcodes	length	parameters
R_NO_RELOCATION	0-23	1	L = (D+1) * 4
	24-27	2	L = (D<<8 + B1 + 1) * 4
	28-30	3	L = (D<<16 + B2 + 1) * 4
	31	4	L = B3 + 1
R_ZEROES	32	2	L = (B1 + 1) * 4
	33	4	L = B3 + 1
R_UNINIT	34	2	L = (B1 + 1) * 4
	35	4	L = B3 + 1
R_RELOCATION	36	1	none
R_DATA_ONE_SYMBOL	37	2	S = B1
	38	4	S = B3
R_DATA_PLABEL	39	2	S = B1
	40	4	S = B3
R_SPACE_REF	41	1	none
R_REPEATED_INIT	42	2	L = 4; M = (B1 + 1) * 4
	43	3	L = (B1 + 1) * 4; M = (B1 + 1) * L
	44	5	L = (B1 + 1) * 4; M = (B3 + 1) * 4
	45	8	L = B3 + 1; M = B4 + 1

Table 10-9: Fixup Request Opcodes and Parameters

mnemonic	opcodes	length	parameters
R_PCREL_CALL	48-57	2	R = rbits1(D); S = B1
	58-59	3	R = rbits2(D<<8 + B1); S = B1
	60-61	5	R = rbits2(D<<8 + B1); S = B3
R_ABS_CALL	64-73	2	R = rbits1(D); S = B1
	74-75	3	R = rbits2(D<<8 + B1); S = B1
	76-77	5	R = rbits2(D<<8 + B1); S = B3
R_DP_RELATIVE	80-111	1	S = D
	112	2	S = B1
	113	4	S = B3
R_DLT_REL	120	2	S = B1
	121	4	S = B3
R_CODE_ONE_SYMBOL	128-159	1	S = D
	160	2	S = B1
	161	4	S = B3
R_MILLI_REL	174	2	S = B1
	175	4	S = B3
R_CODE_PLABEL	176	2	S = B1
	177	4	S = B3
R_BREAKPOINT	178	1	none
R_ENTRY	179	9	U,F = B8 (U is 37 bits; F is 27 bits)
	180	6	U = B5 >> 3; F = pop A
R_ALT_ENTRY	181	1	none
R_EXIT	182	1	none
R_BEGIN_TRY	183	1	none
R_END_TRY	184	1	R = 0
	185	2	R = B1 * 4
	186	4	R = sign-extend(B3) * 4
R_BEGIN_BRTAB	187	1	none
R_END_BRTAB	188	1	none
R_STATEMENT	189	2	N = B1
	190	3	N = B2

Table 10-9: Fixup Request Opcodes and Parameters

mnemonic	opcodes	length	parameters
	191	4	N = B3
R_DATA_EXPR	192	1	none
R_CODE_EXPR	193	1	none
R_FSEL	194	1	none
R_LSEL	195	1	none
R_RSEL	196	1	none
R_N_MODE	197	1	none
R_S_MODE	198	1	none
R_D_MODE	199	1	none
R_R_MODE	200	1	none
R_DATA_OVERRIDE	201	1	V = 0
	202	2	V = sign-extend(B1)
	203	3	V = sign-extend(B2)
	204	4	V = sign-extend(B3)
	205	5	V = B4
R_AUX_UNWIND	207	12	CU,SN,SK = B11 (CU is 24 bits;SN is 32 bitd)
R_COMP1	208	2	OP = B1; V = OP & 0x3f; C = OP & 0x1f
R_COMP2	209	5	OP = B1; S = B3; V = ((OP & 0x7f) << 24) | S
R_COMP3	210	6	OP = B1; V = B4; R = ((OP & 1) << 8) | (V >> 16); S = V & 0xffffff
R_PREV_FIXUP	211-214	1	X = D
R_SEC_STMT	215	1	none
R_RESERVED	224-255	-	reserved

Table 10-10: Fixup Request Secondary Opcodes

Fixup Request	Secondary Mnemonic	Secondary Opcode
R_COMP1	R_PUSH_PCON1	0x00 - 0x3f
	R_PUSH_DOT	0x40
	R_MAX	0x41
	R_MIN	0x42
	R_ADD	0x43
	R_SUB	0x44
	R_MULT	0x45
	R_DIV	0x46
	R_MOD	0x47
	R_AND	0x48
	R_OR	0x49
	R_XOR	0x4a
	R_NOT	0x4b
	R_LSHIFT	0x60 - 0x7f
	R_ARITH_RSHIFT	0x80 - 0x9f
	R_LOGIC_RSHIFT	0xa0 - 0xbf
	R_PUSH_NCON1	0xc0 - 0xff
R_COMP2	R_PUSH_PCON2	0x00
	R_PUSH_SYM	0x80
	R_PUSH_PLABEL	0x82
	R_PUSH_NCON2	0xc0 - 0xff
R_COMP3	R_PUSH_PROC	0x00 - 0x01
	R_PUSH_CONST	0x02

Relocatable Library File Definition

A relocatable library is a file of one or more SOMs and the data structures needed to efficiently manage the SOMs embedded in an archive format. 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, the import table, an optional area for auxiliary headers and the free space list.

In the archive format, the file begins with the archive *magic string*, which consists of the eight ASCII characters "!<arch>\n" (where '\n' refers to the newline or line feed character, hex 0A). Following The magic string are a series of archive headers that describe the section of the library that follows.

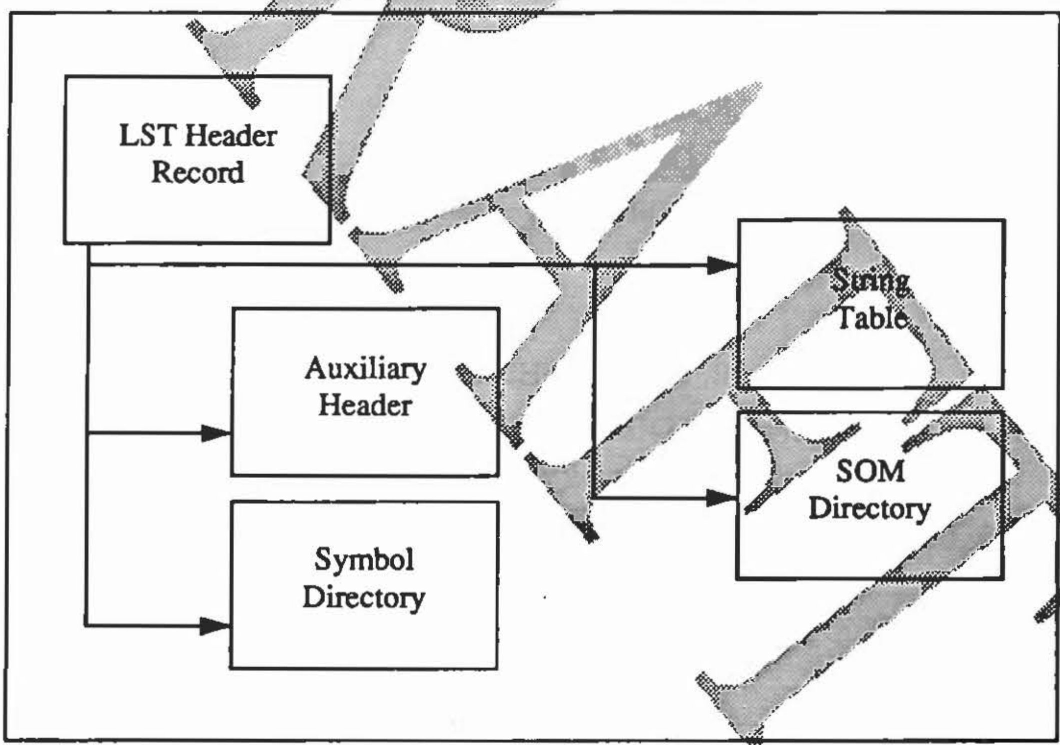

Figure 10-7: Block Diagram of an LST

In all relocatable libraries, the first archive section is the library symbol table (or LST). The LST will have a zero length name (*ar_name* will consist of a single slash ('/') padded with blanks). The remaining archive sections each contain a single SOM.

If the archive contains SOMs with names of a length greater than 15 characters, the archive will contain a string table as the second archive section. The string table will also have a zero length name, but it will consist of two slashes ("//") padded with blanks.

The string table entries will consist of the full length ASCII filename, terminated by a slash followed by a newline character. The archive header for a SOM using the string table will consist of a '/', followed by the string table offset (in ASCII decimal) This offset does not include the preceding archive header. For the example in Figure 10-8, the SOM thisisaverylongfilename.o will have an *ar_name* of "/0" while the SOM thisisalsolong.o will have an *ar_name* of "/27".

Offset	+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	t	h	i
30	s	i	s	a	l	s	o	l	o	n
40	g	[illegible]	o	/	\n					

Figure 10-8: Example of an Archive String Table

Archive Header

The archive header appears in front of every SOM 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 archiver utility.

All of the information in the archive header is in printable ascii format.

.

```
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 10-9: Definition of Archive Header Record

Archive Header Fields

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.

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_fmag

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

LST Header

The Library Symbol Table always begins with a LST header record. For a relocatable library, the LST 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, an optional area for auxiliary headers and the free space list.

```
struct lst_header {
    short int       system_id;
    short int       a_magic;
    unsigned int    version_id;
    unsigned in     file_time[2];
    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    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 10-10: LST Header Definition

LST Header Fields

system_id

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). Both system_id and a_magic are common to the LST and the SOM header. See "system_id" on page 5-9.

a_magic

This is a number that indicates the format and function of the file, for a relocatable library the expected value is 0619 (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

The file time is a 64 bit value that represents the time the file was last modified. The file time is composed of two 32 bit quantities. 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.

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 field is reserved. It must be set to zero by a conforming application.

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. See "Symbol Dictionary" on page 10-20.

string_size

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

free_list

This field is reserved. It must be set to zero by a conforming application.

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, are undefined bits in this record they must be set to zero so that they do not affect the value of *checksum*.

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.

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 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 in length, 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).

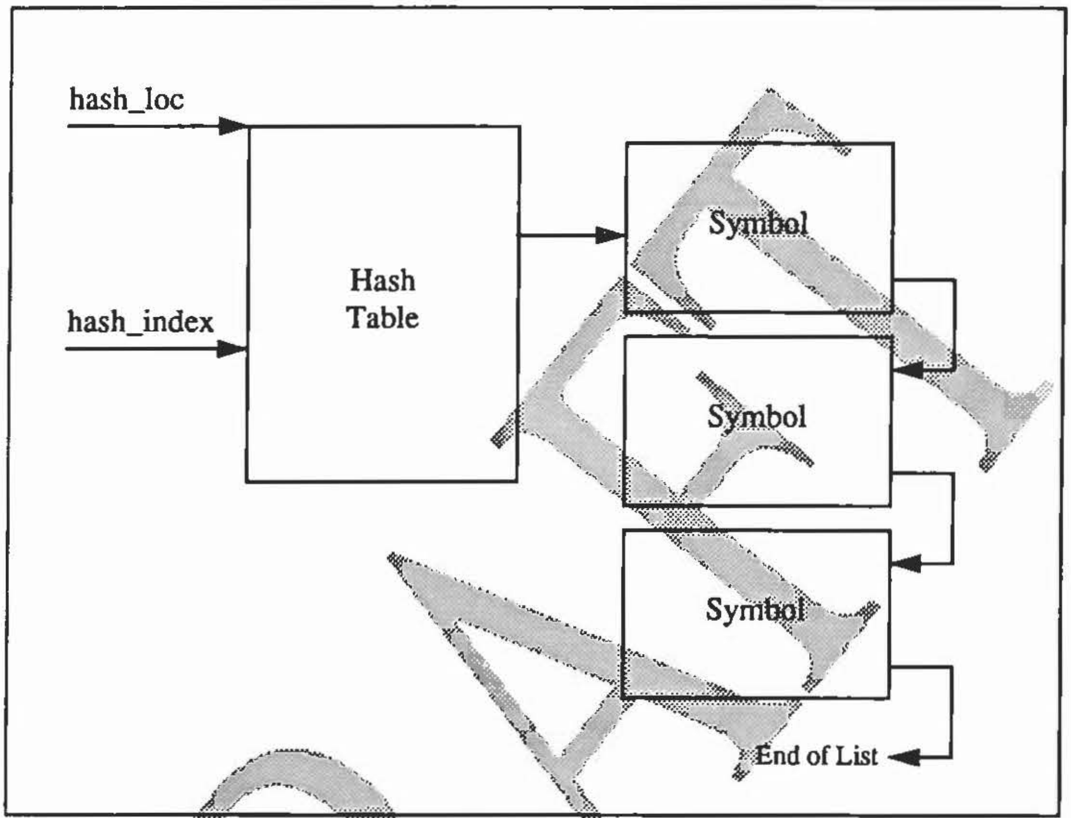

Figure 10-11: Block Diagram of Symbol Directory

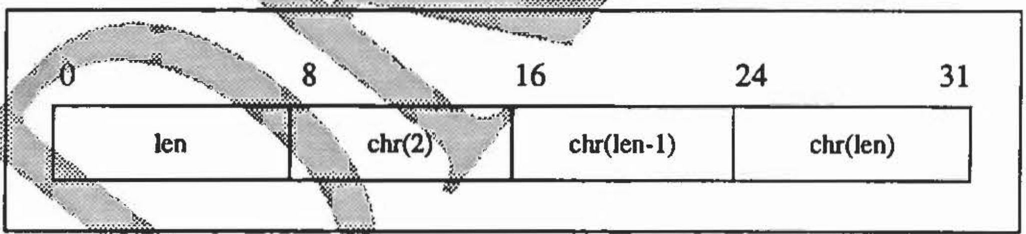

Figure 10-12: *hash_key* Format (symbol length > 1 byte)

0	8	16	24	31
len (=1)	chr(1)	len (=1)	chr(1)	

Figure 10-13: ***hash_key*** **Format (symbol length = 1 byte)**

LST Symbol Record

A symbol record consists of a symbol header record and 0 to 255 argument descriptors constructed as shown in Figure 10-14.

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

```
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;
    unsigned int    name;
    unsigned int    qualifier_name;
    unsigned int    symbol_info;
    unsigned int    symbol_value;
    unsigned int    symbol_descriptor;
    unsigned int    reserved4;
    unsigned int    som_index;
    unsigned int    symbol_key;
    unsigned int    next_entry;
};
```

Figure 10-14: LST Symbol Record Definition

LST Symbol Record Fields

hidden

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

If this flag is set to one, the symbol is a secondary definition and has an additional name (or synonym). The linker will ignore duplicate definitions involving secondary definitions. See "Synonyms" on page 7-3.

symbol_type

This field defines what type of information this symbol represents.
See "symbol_type" on page 10-21 for more information.

symbol_scope

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_scope" on page 10-23.

check_level

This field is reserved. It must be set to 0 in conforming applications.

must_qualify

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

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

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

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

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

This field is undefined by the PRO ABI. Conforming applications and systems should not interpret the contents of this field.

arg_reloc

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 "arg_reloc" on page 10-25.

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 in the 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_info" on page 10-27.

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_value" on page 10-27.

symbol_descriptor

This field is reserved. It shall be initialized to 0.

reserved

This field is reserved. It shall be initialized to 0.

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 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 Dictionary" on page 10-20 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.

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 *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 10-15 shows the structure of the SOM directory.

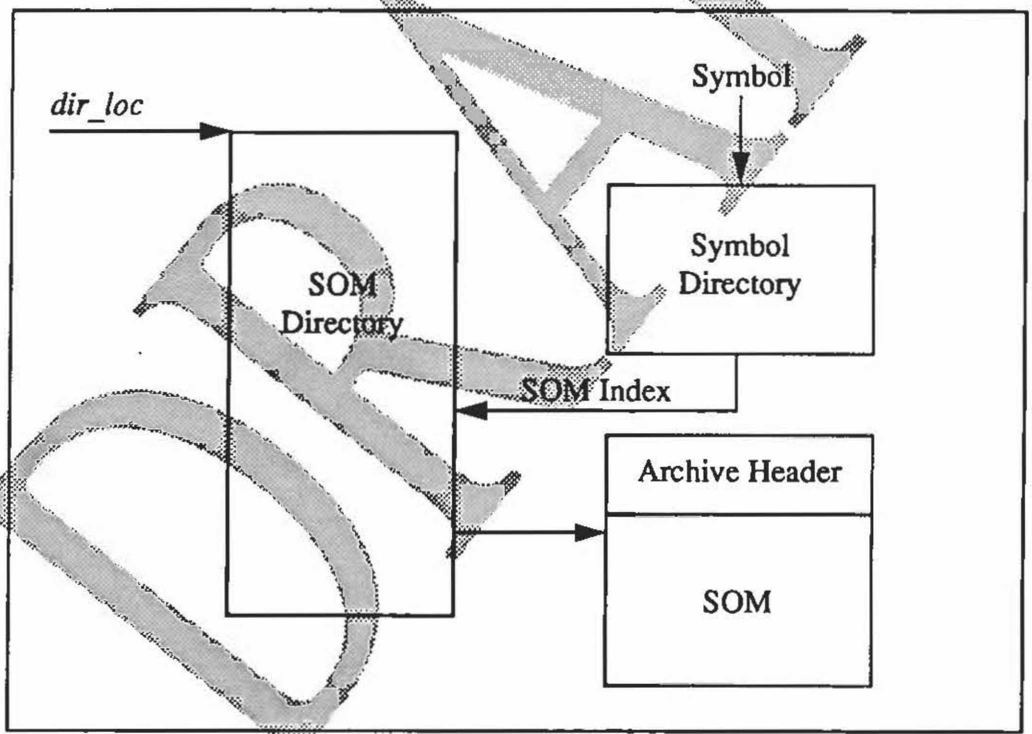

Figure 10-15: Structure of the SOM directory

SOM Directory Entry

Each entry in the SOM directory has the format indicated in Figure 10-16.

```
struct som_entry {
        unsigned int    location;
        unsigned int    length;
};
```

Figure 10-16: Definition of a SOM Directory Entry

Parameter Relocation

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

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

The compiler or assembler must communicate the location of the first four words of the parameter list and the location of the function return value to the linker and loader. To accomplish this, ten bits of argument location information have been added to the definitions of a symbol and a fix-up request. The following diagram shows the first word of a symbol dictionary record in the object file. See "Symbol Dictionary" on page 10-20.

bits →	8	4	4	2	1	1	1	1	10
	symbol type	scope	check	XL	Q	N	M	R	argument location

Figure 10-17: 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)[1]
11	frupper	Floating-point register (bits 32..63)[1]

1. 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.

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 10-18.

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. import 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".)

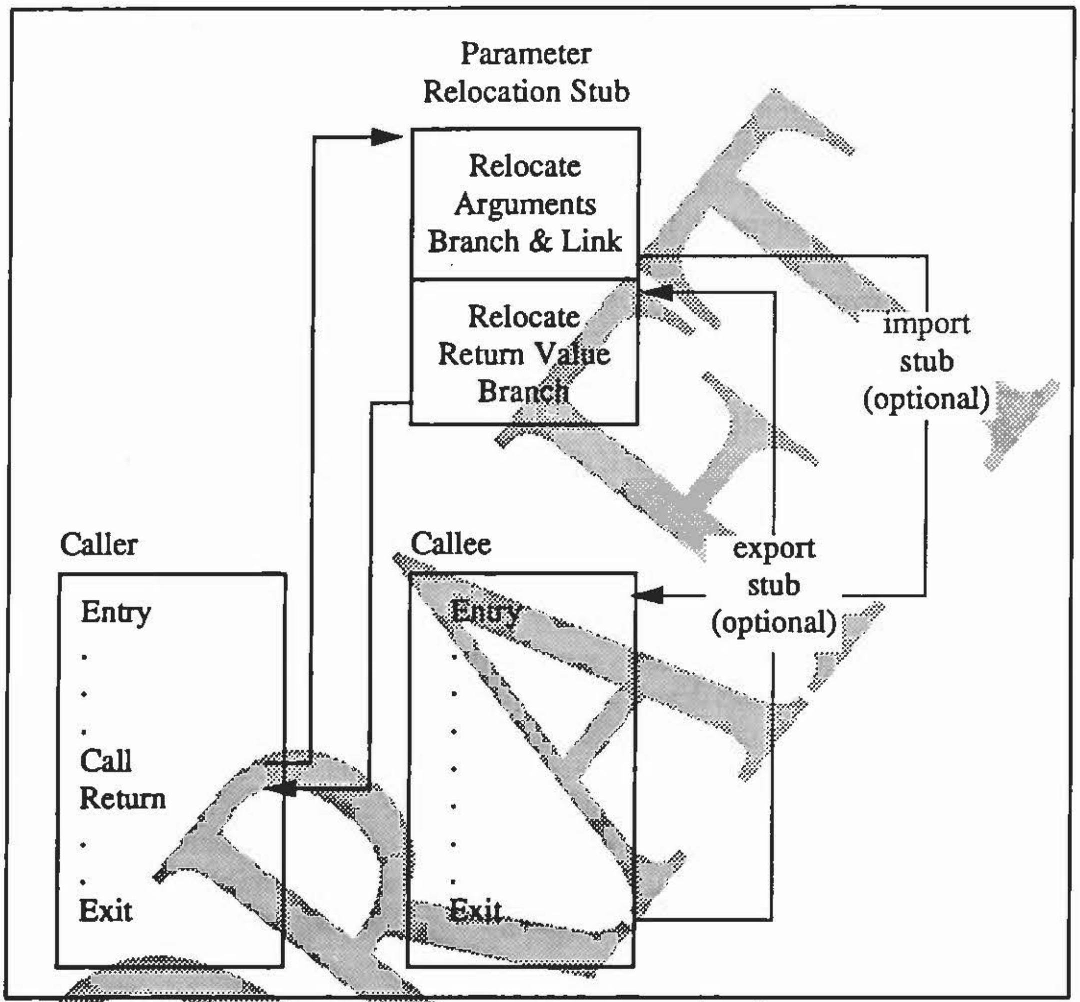

Figure 10-18: Parameter Relocation Stub.

When the linker makes a load module executable, it will generate stubs for each procedure that can be called from another load module (i.e. called dynamically). In addition, a stub will be required for each possible calling sequence. Each of these stubs will contain the code for both relocation and external return, and will be required to contain a symbol definition record. Both import and export stubs use a standard interface: import stubs always relocate arguments to general registers, and export stubs always assume general registers. See "External Calls" on page 4-19.

Millicode Calls

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

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

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

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

The PRO ABI does not require that any specific millicode libraries are present on a conforming system. For an application to ship as a conforming object file, any millicode libraries required by the object must ship as part of the application.

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.

Note

If a procedure only makes millicode calls, it may be considered a leaf routine. See "Leaf and Non-Leaf Procedures" on page 4-2.

Making a Millicode Call

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

- The return address (MRP) is passed in GR 31; and
- Function results are returned in GR 29.

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

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.

A

Glossary

ABI

Application Binary Interface.

ANSI

American National Standards Institute.

API

Application Programming Interface.

PA-RISC

The processor architecture defined by the "PA-RISC 1.1 Architecture and Instruction Set Reference Manual".

PRO

The Precision Risc Organization.

SOM

Standard Object Module.

absolute path name

An absolute path name is a fully qualified file system path that begins at "/".

archive format

The format used in relocatable libraries to separate the LST header and individual SOMs.

binding

The process of resolving unsatisfied symbols in a SOM to symbols exported by other SOMs in a library or relocatable object.

called stub

see export stub.

callee-saves register

The value in a callee-save register must be saved by a procedure immediately after the procedure is entered, and restored immediately before the procedure exits, if it is used by the current procedure. The values in the callee-saves registers are guaranteed to be preserved across procedure calls.

caller-saves register

The value in a caller-saves register are not saved across procedure calls. Any procedure may use caller-saves registers at any time. If a procedure requires a value in a caller-save register to persist across a procedure call, it must save and restore the value itself.

calling stub

see import stub.

complete executable

An executable that has no external dependencies, usually statically linked.

conforming application

An application that uses only the functionality allowed by the PRO ABI and API and has been certified by the PRO conformance tests.

conforming system

A system that provides all of the functionality specified by the PRO ABI and API and has been certified by the PRO conformance tests.

data linkage table

A table that is initialized by the dynamic loader which contains a one word entry for each data item that is referenced by PIC.

dynamic library

A library that is bound to an incomplete executable by the dynamic loader. A dynamic library is named with the ".sl" suffix.

dynamic loader

A dynamic library, dld.sl, that is called by an incomplete executables start-up code and binds all unresolved data and procedure references.

exception

An interruption of program execution due to exceptional conditions that must be resolved before execution may continue.

export stub

A stub used to trap the return from a routine and perform the required inter-space branch between a dynamic library and an incomplete executable. Also known as a called stub.

fixup

A request provided to the linker regarding the placement of data in the resultant executable.

frame marker

An eight word area at the top of a stack frame that contains storage areas for values used by stubs.

gateway page

The first page of the fourth quadrant. It contains the interface between the executable and the kernel for use with system calls. This interface is only to be used to allow the start-up code to access the dynamic loader.

high water mark

A version identifier used in dynamic libraries to mark an incompatible change. When binding to a dynamic library, an incomplete executable matches to a high water mark that is less than or equal that found when the executable was linked.

import stub

A stub used for making an external call. It calls the export stub of the desired procedure after saving the information the export stub will need to return to the calling code. Also known as a calling stub.

incomplete executable

An executable that contains unsatisfied references to external code and/or data provided in a dynamic library. These external references are resolved by the dynamic loader when the incomplete executable is executed.

initializer

An initializer is a routine that is called whenever a dynamic library is loaded or unloaded.

leaf procedure

A procedure that does not make any procedure calls.

library search path

A directory list that may be searched by the dynamic loader to find any dynamic libraries that are referenced by the incomplete executable.

linkage table

A branch table used to handle indirect procedure and data references.

linkage table pointer

A pointer into the linkage table used with import stubs in a dynamic library. The LTP is stored in GR 19 by convention.

linker

A utility that combines relocatable objects and libraries into a single (usually executable) file.

loop

See recursion.

millicode

A special function calling convention with less overhead than standard calls, or code accessed using the millicode calling convention.

millicode return pointer

The return address used in a millicode call. Stored in GR 31.

non-leaf procedure

A procedure that makes calls to other procedures before exiting.

parameter relocation

The rearrangement of parameters during a procedure call required when the caller and callee do not agree upon their proper location.

parameter relocation stub

A stub inserted between a caller and callee to perform parameter relocation.

plabel

A 32-bit procedure label that defines a procedure address and space.

position independent code

Code that does not contain references to absolute addresses. All addresses are relative to the program counter, or to a linkage table pointer.

procedure linkage table

A linkage table found in incomplete executables and dynamic libraries that stores the addresses of all unresolved external procedures. It is initialized by the dynamic loader.

quad precision

A floating-point format of 128 bits, or twice the size of double precision.

quadrant

An area of virtual memory one gigabyte (2^{30} bytes) in size and alignment.

recursion

See loop.

relocatable library

A library of SOMs embedded in the archive format. Relocatable libraries, also known as archive libraries, use the ".a" suffix.

relocatable object

A SOM that is in a linkable form. A relocatable object is named with the ".o" suffix.

return pointer

The address to which a procedure shall return to upon exit. Stored in GR2 or on the stack.

shared library

A library that has a single image in memory that is shared by all applications. Most dynamic libraries are also shared libraries.

signal

A message passed between processes. A signal may also be viewed as an event to which processes respond.

sort key

A value used to control the order of spaces and subspaces in memory.

space

A region of virtual memory of four gigabytes (2^{32} bytes).

space id

A unique identifier for a single space used with the space registers.

stack

A region of virtual memory used by an application for data storage and managed by software conventions.

stack frame

The stack is accessed in pieces that are an integer multiple of 64 bytes in size and aligned on a 64 byte boundary. These pieces are called stack frames.

stack pointer

The stack grows towards higher memory addresses with contiguous stack frames. The stack pointer (SP or GR 30) holds the address of the top of the current stack frame.

stub

A short code segment inserted into a procedure calling sequence by a linker and used for a specific purpose such as parameter relocation or inter-space procedure call.

subspace

A logical partition of a space used to group code and data into separate segments.

symbol

The name by which a procedure or data item is referred to in a SOM.

synonym

A second name provided for a symbol. Usually used to provide a clean name space.

system call

A call to the kernel through the Gateway page.

virtual address

An address consisting of a space identifier and 32-bit offset that is independent of location in real memory.

B

Data Structures and Constants

This appendix specifies the PA-RISC implementation for the system data structures. They are defined using ANSI C, as descriibed in chapter 3, as a convenient notation. These headers are listed to describe the data structures and constants used by the system libraries on a conforming system. Conforming systems are not required to provide these headers.

Any structure elements specified as "reserved" by the PRO ABI are reserved for system use. These elements do not have defined semantics and may not be used by conforming applications. They must be initialized to zero unless stated otherwise.

```
extern void _assert(char *, char *, int);

#define  assert(_EX)                                        \
        ((_EX) ? (void)0 : _assert(#_EX, __FILE__, __LINE__))
```

Figure B-1: <assert.h>

```
#define  _tolower(__c)        tolower(__c)
#define  _toupper(__c)        toupper(__c)

#define  toascii(__c)         ((__c) & 0x7f)
```

Figure B-2: <ctype.h>

```
typedef unsigned long         ino_t;

#define  _MAXNAMLEN           255

struct dirent {
    ino_t    d_ino;
    short    d_reclen;
    short    d_namlen;
    char     d_name[_MAXNAMLEN + 1];
};
typedef void *DIR;
```

Figure B-3: <dirent.h>

```
#define  BIND_IMMEDIATE       0x0
#define  BIND_DEFERRED        0x1
#define  BIND_REFERENCE       0x2
#define  BIND_FIRST           0x4
#define  BIND_NONFATAL        0x8
#define  BIND_NOSTART         0x10
#define  BIND_VERBOSE         0x20
#define  BIND_RESTRICTED      0x40
#define  BIND_RESERVED1       0x80

#define  DYNAMIC_PATH         0x100

typedef  void *shl_t;
extern   int __text_start;

#define  PROG_HANDLE          ((shl_t)(&__text_start))

#define MAXPATHLEN 1024
```

Figure 7-4: <dl.h>

```
struct shl_descriptor {
   unsigned long  tstart;
   unsigned long  tend;
   unsigned long  dstart;
   unsigned long  dend;
   void *ltptr;
   shl_t handle;
   char filename[MAXPATHLEN+1];
   void *initializer;
   unsigned long  ref_count;
};

struct shl_symbol {
   char *name;
   short type;
   void *value;
   shl_t handle;
};

#define  TYPE_UNDEFINED        0
#define  TYPE_DATA             2
#define  TYPE_PROCEDURE        3
#define  TYPE_STORAGE          7

#define  IMPORT_SYMBOLS        0x01
#define  EXPORT_SYMBOLS        0x02
#define  NO_VALUES             0x04
#define  GLOBAL_VALUES         0x08
#define  INITIALIZERS          0x10
```

Figure 7-4: <dl.h>

```
#define  EPERM        1
#define  ENOENT       2
#define  ESRCH        3
#define  EINTR        4
#define  EIO          5
#define  ENXIO        6
#define  E2BIG        7
#define  ENOEXEC      8
#define  EBADF        9
#define  ECHILD       10
#define  EAGAIN       11
#define  ENOMEM       12
#define  EACCES       13
#define  EFAULT       14
#define  ENOTBLK      15
#define  EBUSY        16
#define  EEXIST       17
#define  EXDEV        18
#define  ENODEV       19
#define  ENOTDIR      20
#define  EISDIR       21
#define  EINVAL       22
#define  ENFILE       23
#define  EMFILE       24
#define  ENOTTY       25
#define  ETXTBSY      26
#define  EFBIG        27
#define  ENOSPC       28
#define  ESPIPE       29
#define  EROFS        30
#define  EMLINK       31
#define  EPIPE        32
#define  EDOM         33
#define  ERANGE       34
#define  ENOMSG       35
#define  EIDRM        36
#define  EDEADLK      45
#define  ENOLCK       46
#define  EILSEQ       47
```


Figure B-5: <errno.h>

```
#define   ENOSYM            215
#define   ENOTSOCK          216
#define   EDESTADDRREQ      217
#define   EMSGSIZE          218
#define   EPROTOTYPE        219
#define   ENOPROTOOPT       220
#define   EPROTONOSUPPORT   221
#define   ESOCKTNOSUPPORT   222
#define   EOPNOTSUPP        223
#define   EPFNOSUPPORT      224
#define   EAFNOSUPPORT      225
#define   EADDRINUSE        226
#define   EADDRNOTAVAIL     227
#define   ENETDOWN          228
#define   ENETUNREACH       229
#define   ENETRESET         230
#define   ECONNABORTED      231
#define   ECONNRESET        232
#define   ENOBUFS           233
#define   EISCONN           234
#define   ENOTCONN          235
#define   ESHUTDOWN         236
#define   ETOOMANYREFS      237
#define   ETIMEDOUT         238
#define   ECONNREFUSED      239
#define   EREFUSED          ECONNREFUSED
#define   EREMOTERELEASE    240
#define   EHOSTDOWN         241
#define   EHOSTUNREACH      242
#define   EALREADY          244
#define   EINPROGRESS       245
#define   EWOULDBLOCK       246
#define   ENOTEMPTY         247
#define   ENAMETOOLONG      248
#define   ELOOP             249
#define   ENOSYS            251

extern int errno;
```

Figure B-5: <errno.h>

```
#define  F_DUPFD            0
#define  F_GETFD            1
#define  F_SETFD            2
#define  F_GETFL            3
#define  F_SETFL            4
#define  F_GETLK            5
#define  F_SETLK            6
#define  F_SETLKW           7

#define  FD_CLOEXEC         1

#define  F_RDLCK            1
#define  F_WRLCK            2
#define  F_UNLCK            3

#define  O_RDONLY           0000000
#define  O_WRONLY           0000001
#define  O_RDWR             0000002
#define  O_ACCMODE          0000003
#define  O_NDELAY           0000004
#define  O_APPEND           0000010
#define  O_CREAT            0000400
#define  O_TRUNC            0001000
#define  O_EXCL             0002000
#define  O_SYNC             0100000
#define  O_SYNCIO           O_SYNC
#define  O_NONBLOCK         0200000
#define  O_NOCTTY           0400000

struct flock {
    short  l_type;
    short  l_whence;
    off_t  l_start;
    off_t  l_len;
    pid_t  l_pid;
};
```

Figure B-6: <fcntl.h>

```
#define  FLT_RADIX        2
#define  FLT_ROUNDS       1

/*       FLT_EPSILON      0x34000000
         FLT_MIN          0x00800000
         FLT_MAX          0x7f7fffff
*/

#define  FLT_MANT_DIG     24
#define  FLT_EPSILON      1.19209290E-07F
#define  FLT_DIG          6
#define  FLT_MIN_EXP      (-125)
#define  FLT_MIN          1.17549435E-38F
#define  FLT_MIN_10_EXP (-37)
#define  FLT_MAX_EXP      128
#define  FLT_MAX          3.40282347E+38F
#define  FLT_MAX_10_EXP 38

/*       DBL_EPSILON      0x3cb00000 0x00000000
         DBL_MIN          0x00100000 0x00000000
         DBL_MAX          0x7fefffff 0xffffffff
*/

#define  DBL_MANT_DIG     53
#define  DBL_EPSILON      2.2204460492503131E-16
#define  DBL_DIG          15
#define  DBL_MIN_EXP      (-1021)
#define  DBL_MIN          2.2250738585072014E-308
#define  DBL_MIN_10_EXP (-307)
#define  DBL_MAX_EXP      1024
#define  DBL_MAX          1.7976931348623157E+308
#define  DBL_MAX_10_EXP 308

/* LDBL_EPSILON    0x3f8f0000 0x00000000 0x00000000 0x00000000
   LDBL_MIN        0x00010000 0x00000000 0x00000000 0x00000000
   LDBL_MAX        0x7ffeffff 0xffffffff 0xffffffff 0xffffffff
*/

#define LDBL_MANT_DIG  113
#define LDBL_EPSILON  1.925929944387235853055977942584927 3E-34L
```

Figure B-7: <float.h>

```
#define  LDBL_DIG        33
#define  LDBL_MIN_EXP    (-16381)
#define  LDBL_MIN
         3.3621031431120935062626778173217526026E-4932L
#define  LDBL_MIN_10_EXP  (-4931)
#define  LDBL_MAX_EXP    16384
#define  LDBL_MAX
         1.1897314953572317650857593266280070162E4932L
#define  LDBL_MAX_10_EXP  4932
```

Figure B-7: <float.h>

```
#define  FNM_PATHNAME       0x01
#define  FNM_PERIOD         0x02
#define  FNM_NOESCAPE       0x08

#define  FNM_NOMATCH        1
#define  FNM_NOSYS          2
```

Figure B-8: <fnmatch.h>

```
#define  FTW_F              0
#define  FTW_D              1
#define  FTW_DNR            2
#define  FTW_NS             3
#define  FTW_DP             4
#define  FTW_SL             5

#define  FTW_PHYS           1
#define  FTW_MOUNT          2
#define  FTW_DEPTH          4
#define  FTW_CHDIR          8
#define  FTW_SERR           32

struct FTW {
   int base;
   int level;
};
```

Figure B-9: <ftw.h>

```
typedef struct {
   size_t   gl_pathc;
   char     **gl_pathv;
   size_t   gl_offs;
   char     *gl_mem;
} glob_t;

#define  GLOB_ERR              0x01
#define  GLOB_MARK             0x02
#define  GLOB_NOSORT           0x04
#define  GLOB_NOCHECK          0x08
#define  GLOB_DOOFFS           0x10
#define  GLOB_APPEND           0x20
#define  GLOB_NOESCAPE         0x40

#define  GLOB_NOSPACE          1
#define  GLOB_ABORTED          2
#define  GLOB_NOMATCH          3
#define  GLOB_NOSYS            4
```

Figure B-10: <glob.h>

```
struct group {
   char     *gr_name;
   char     *gr_passwd;
   gid_t    gr_gid;
   char     **gr_mem;
};
```

Figure B-11: <grp.h>

```
typedef int iconv_t;
```

Figure B-12: <iconv.h>

```
#define  D_T_FMT        1
#define  D_FMT          2
#define  T_FMT          3

#define  DAY_1          6
#define  DAY_2          7
#define  DAY_3          8
#define  DAY_4          9
#define  DAY_5          10
#define  DAY_6          11
#define  DAY_7          12

#define  ABDAY_1        13
#define  ABDAY_2        14
#define  ABDAY_3        15
#define  ABDAY_4        16
#define  ABDAY_5        17
#define  ABDAY_6        18
#define  ABDAY_7        19

#define  MON_1          20
#define  MON_2          21
#define  MON_3          22
#define  MON_4          23
#define  MON_5          24
#define  MON_6          25
#define  MON_7          26
#define  MON_8          27
#define  MON_9          28
#define  MON_10         29
#define  MON_11         30
#define  MON_12         31

#define  ABMON_1        32
#define  ABMON_2        33
#define  ABMON_3        34
#define  ABMON_4        35
#define  ABMON_5        36
#define  ABMON_6        37
```

Figure B-13: <langinfo.h>

```
#define  ABMON_7            38
#define  ABMON_8            39
#define  ABMON_9            40
#define  ABMON_10           41
#define  ABMON_11           42
#define  ABMON_12           43

#define  RADIXCHAR          44
#define  THOUSEP            45
#define  YESSTR             46
#define  NOSTR              47
#define  CRNCYSTR           48
#define  BYTES_CHAR         49
#define  DIRECTION          50
#define  ALT_DIGIT          51
#define  ALT_PUNCT          52
#define  AM_STR             53
#define  PM_STR             54
#define  YEAR_UNIT          55
#define  MON_UNIT           56
#define  DAY_UNIT           57
#define  HOUR_UNIT          58
#define  MIN_UNIT           59
#define  SEC_UNIT           60
#define  ERA_D_FMT          61      /* New values     */
#define  CODESET            62      /* expected       */
#define  YESEXPR            72
#define  NOEXPR             73
#define  T_FMT_AMPM         74
#define  ALT_DIGITS         75
#define  ERA                76
#define  ERA_D_T_FMT        77
#define  ERA_T_FMT          78

typedef int nl_item;
extern char *nl_langinfo(nl_item);
```

Figure B-13: <langinfo.h>

```
#undef   ARG_MAX
#undef   BC_BASE_MAX
#undef   BC_DIM_MAX
#undef   BC_SCALE_MAX
#undef   BC_STRING_MAX
#undef   CHILD_MAX
#undef   COLL_WEIGHTS_MAX
#undef   EXPR_NEST_MAX
#undef   LINE_MAX
#undef   NGROUPS_MAX
#undef   OPEN_MAX
#undef   PASS_MAX
#undef   RE_DUP_MAX
#undef   STREAM_MAX
#undef   TZNAME_MAX

#define  _POSIX_ARG_MAX              4096
#define  _POSIX_CHILD_MAX            6
#define  _POSIX_LINK_MAX             8
#define  _POSIX_MAX_CANON            255
#define  _POSIX_MAX_INPUT            255
#define  _POSIX_NAME_MAX             14
#define  _POSIX_NGROUPS_MAX          0
#define  _POSIX_OPEN_MAX             16
#define  _POSIX_PATH_MAX             255
#define  _POSIX_PIPE_BUF             512
#define  _POSIX_SSIZE_MAX            32767
#define  _POSIX_STREAM_MAX           8
#define  _POSIX_TZNAME_MAX           3

#define  _POSIX2_BC_BASE_MAX         99
#define  _POSIX2_BC_DIM_MAX          2048
#define  _POSIX2_BC_SCALE_MAX        99
#define  _POSIX2_BC_STRING_MAX       1000
#define  _POSIX2_COLL_WEIGHTS_MAX    2
#define  _POSIX2_EXPR_NEST_MAX       32
#define  _POSIX2_LINE_MAX            2048
#define  _POSIX2_RE_DUP_MAX          255

#define  LINK_MAX                    32767
```

Figure B-14: <limits.h>

```
#define  MAX_CANON               512
#define  MAX_INPUT               512
#define  NAME_MAX                14
#define  PATH_MAX                1023
#define  PIPE_BUF                8192

#define  CHAR_BIT                8
#define  CHAR_MAX                127
#define  DBL_DIG                 15
#define  DBL_MAX                 1.7976931348623157e+308
#define  FLT_DIG                 6
#define  FLT_MAX                 3.40282347e+38
#define  INT_MAX                 2147483647
#define  LONG_BIT                32
#define  LONG_MAX                2147483647L
#define  MB_LEN_MAX              4
#define  SCHAR_MAX               127
#define  SHRT_MAX                32767
#define  SSIZE_MAX               INT_MAX
#define  UCHAR_MAX               255
#define  UINT_MAX                4294967295U
#define  ULONG_MAX               4294967295UL
#define  USHRT_MAX               65535
#define  WORD_BIT                32

#define  CHAR_MIN                (-128)
#define  INT_MIN                 (-2147483647 - 1)
#define  LONG_MIN                (-2147483647L - 1)
#define  SCHAR_MIN               (-128)
#define  SHRT_MIN                (-32768)

#define  CHARCLASS_NAME_MAX      14
#define  NL_ARGMAX               9
#define  NL_LANGMAX              44
#define  NL_MSGMAX               65534
#define  NL_NMAX                 2
#define  NL_SETMAX               255
#define  NL_TEXTMAX              8192
#define  NZERO                   20
#define  TMP_MAX                 17576
```

Figure B-14: <limits.h>

```
#define  LC_ALL             0
#define  LC_COLLATE         1
#define  LC_CTYPE           2
#define  LC_MONETARY        3
#define  LC_NUMERIC         4
#define  LC_TIME            5
#define  LC_MESSAGES        6
#define  LOCALE_STATUS      1
#define  MODIFIER_STATUS    2
#define  ERROR_STATUS       3

struct lconv {
   char  *decimal_point;
   char  *thousands_sep;
   char  *grouping;
   char  *int_curr_symbol;
   char  *currency_symbol;
   char  *mon_decimal_point;
   char  *mon_thousands_sep;
   char  *mon_grouping;
   char  *positive_sign;
   char  *negative_sign;
   char  int_frac_digits;
   char  frac_digits;
   char  p_cs_precedes;
   char  p_sep_by_space;
   char  n_cs_precedes;
   char  n_sep_by_space;
   char  p_sign_posn;
   char  n_sign_posn;
   };
struct locale_data {
   char  LC_ALL_D[59];
   char  LC_COLLATE_D[59];
   char  LC_CTYPE_D[59];
   char  LC_MONETARY_D[59];
   char  LC_NUMERIC_D[59];
   char  LC_TIME_D[59];
   char  LC_MESSAGES_D[59];
};
```

Figure B-15: <locale.h>

```
            /* HUGE_VAL = 0x7fefffff 0xffffffff */
#define  HUGE_VAL              1.7976931348623157e+308
#define  FP_PLUS_NORM          0
#define  FP_MINUS_NORM         1
#define  FP_PLUS_ZERO          2
#define  FP_MINUS_ZERO         3
#define  FP_PLUS_INF           4
#define  FP_MINUS_INF          5
#define  FP_PLUS_DENORM        6
#define  FP_MINUS_DENORM       7
#define  FP_SNAN               8
#define  FP_QNAN               9
#define  FP_X_INV              0x10
#define  FP_X_DZ               0x08
#define  FP_X_OFL              0x04
#define  FP_X_UFL              0x02
#define  FP_X_IMP              0x01
#define  FP_X_CLEAR            0x00

typedef struct {
   unsigned int    word1, word2, word3, word4;
} long_double;

typedef long       fp_control;
typedef int        fp_except;
extern int  signgam;

typedef enum {
   FP_RZ,
   FP_RN,
   FP_RP,
   FP_RM
} fp_rnd;
```

Figure B-16: <math.h>

```
typedef unsigned int     size_t;
typedef int              ssize_t;
```

Figure B-17: <monetary.h>

```
#define  NL_SETD               1
#define  NL_CAT_LOCALE         1

typedef int  nl_catd;
typedef int  nl_item;
```

Figure B-18: <nl_types.h>

```
struct pollfd {
   int fd;
   short events;
   short revents;
};

#define  POLLIN               01
#define  POLLNORM             POLLIN
#define  POLLPRI              02
#define  POLLOUT              04

#define  POLLERR              010
#define  POLLHUP              020
#define  POLLNVAL             040

#define  NPOLLFILE            20
```

Figure B-19: <poll.h>

```
struct passwd {
   char  *pw_name;
   char  *pw_passwd;
   uid_t  pw_uid;
   gid_t  pw_gid;
   char  *pw_age;
   char  *pw_comment;
   char  *pw_gecos;
   char  *pw_dir;
   char  *pw_shell;
};
```

Figure B-20: <pwd.h>

```
#define  REG_NPAREN          255

#define  REG_EXTENDED        0001
#define  REG_NEWLINE         0002
#define  REG_ICASE           0004
#define  REG_NOSUB           0010

#define  REG_NOTBOL          0001
#define  REG_NOTEOL          0002

#define  REG_NOMATCH         20
#define  REG_BADPAT          2
#define  REG_ECOLLATE        21
#define  REG_ECTYPE          24
#define  REG_EESCAPE         22
#define  REG_ESUBREG         25
#define  REG_EBRACK          49
#define  REG_EPAREN          42
#define  REG_EBRACE          45
#define  REG_BADBR           3
#define  REG_ERANGE          23
#define  REG_ESPACE          50
#define  REG_BADRPT          26
#define  REG_ENEWLINE        36
#define  REG_ENSUB           43
#define  REG_EMEM            51
#define  REG_ENOSEARCH       41
#define  REG_EDUPOPER        26
#define  REG_ENOEXPR         27
#define  REG_ENOSYS          28

typedef int regoff_t;

typedef struct {
   regoff_t  rm_so;
   regoff_t  rm_eo;
} regmatch_t;
```

Figure B-21: <regex.h>

```
typedef struct {
   unsigned char *__c_re;
   unsigned char *__c_re_end;
   unsigned char *__c_buf_end;
   size_t re_nsub;
   int __anchor;
   int __flags;
} regex_t;
```

Figure B-21: <regex.h>

```
/* The variables and routines identified in regexp.h must
   be defined local to an application. These routines must
   conform to the PRO Standards. It is recommended that
   applications use the regex interface. */

char *loc1, *loc2, *locs;

char *compile(register char *instring, register char *ep,\
   const char *endbuf, int seof);
step(const char *string, const char *expbuf);
advance(const char *string, const char *expbuf);
```

Figure 7-22: <regexp.h>

```
typedef struct entry { char *key; void *data; } ENTRY;

typedef enum { FIND, ENTER } ACTION;

typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

Figure B-23: <search.h>

```
typedef double jmp_buf[25];
typedef double sigjmp_buf[25];
```

Figure B-24: <setjmp.h>

```
#define DL_HDR_VERSION_ID  89060912
#define SHLIB_UNW_VERS_ID  89081712

#define DLT_ENTRY int

struct dl_header {
   int hdr_version;
   int ltptr_value;
   int shlib_list_loc;
   int shlib_list_count;
   int import_list_loc;
   int import_list_count;
   int hash_table_loc;
   int hash_table_size;
   int export_list_loc;
   int export_list_count;
   int string_table_loc;
   int string_table_size;
   int dreloc_loc;
   int dreloc_count;
   int dlt_loc;
   int plt_loc;
   int dlt_count;
   int plt_count;
   short highwater_mark;
   short flags;
   int export_ext_loc;
   int module_loc;
   int module_count;
   int elaborator;
   int initializer;
   int embedded_path;
   int reserved2;
   int reserved3;
   int reserved4;
};
```

Figure B-25: <shl.h>

```
#define ELAB_DEFINED        0x1
#define INIT_DEFINED        0x2
#define SHLIB_PATH_ENABLE   0x4
#define EMBED_PATH_ENABLE   0x8
#define SHLIB_PATH_FIRST    0x10

struct import_entry {
    int name;
    short reserved1;
    unsigned char type;
    unsigned int bypassable:1;
    unsigned int reserved2:7;
};

struct misc_info {
    short version;
    unsigned int reserved2 : 6;
    unsigned int arg_reloc : 10;
};

struct export_entry {
    int next;
    int name;
    int value;
    union {
        int size;
        struct misc_info misc;
    } info;
    unsigned char type;
    char reserved1;
    short module_index;
};

struct export_entry_ext {
    int size;
    int dreloc;
    int same_list;
    int reserved2;
    int reserved3;
};
```

Figure B-25: <shl.h>

```
struct shlib_list_entry {
   int shlib_name;
   unsigned char  dash_l_reference;
   unsigned char  bind;
   short highwater_mark;
};

struct PLT_entry {
   int proc_addr;
   int ltptr_value;
};

struct dreloc_record {
   int shlib;
   int symbol;
   int location;
   int value;
   unsigned char type;
   char reserved;
   short module_index;
};

#define DR_PLABEL_EXT      1  /* Dynamic Relocation Types */
#define DR_PLABEL_INT      2
#define DR_DATA_EXT        3
#define DR_DATA_INT        4
#define DR_PROPAGATE       5
#define DR_INVOKE          6
#define DR_TEXT_INT        7

struct module_entry {
   int drelocs;
   int imports;
   int import_count;
   char flags;
   char reserved1;
   unsigned short  module_dependencies;
   int reserved2;
};
```

Figure B-25: <shl.h>

```
#define ELAB_REF            0x1

struct dld_parms {
   long version;          /* version num of dld_parms */
   long text_addr;        /* text address of dld */
   long text_end;         /* text end of dld */
   long prog_data_addr;   /* start of data in program file */
   char **envp;           /* environment pointer */
};

struct shlib_unwind_info {
   int magic;
   int shlib_name;
   int text_start;
   int data_start;
   int unwind_start;
   int unwind_end;
   int recover_start;
   int recover_end;
};

#define PARMS_STRUCT_FLD4   0
#define PARMS_STRUCT_FLD5   1
#define PARMS_STRUCT_USED   (-1)
```

Figure B-25: <shl.h>

```
typedef unsigned int  sig_atomic_t;

typedef struct {
   long sigset[8];
} sigset_t;

#define  sv_onstack          sv_flags

struct sigaction {
   void  (*sa_handler)();
   sigset_t  sa_mask;
   int  sa_flags;
};

struct sigstack {
   char  *ss_sp;
   int  ss_onstack;
};

struct sigvec {
   void  (*sv_handler)();
   int  sv_mask;
   int  sv_flags;
};

#define  SIGHUP              1
#define  SIGINT              2
#define  SIGQUIT             3
#define  SIGILL              4
#define  SIGTRAP             5
#define  SIGABRT             6
#define  SIGIOT              SIGABRT
#define  SIGEMT              7
#define  SIGFPE              8
#define  SIGKILL             9
#define  SIGBUS              10
#define  SIGSEGV             11
#define  SIGSYS              12
#define  SIGPIPE             13
#define  SIGALRM             14
```

Figure B-26: <signal.h>

```
#define  SIGTERM            15
#define  SIGUSR1            16
#define  SIGUSR2            17
#define  SIGCHLD            18
#define  SIGCLD             SIGCHLD
#define  SIGPWR             19
#define  SIGVTALRM          20
#define  SIGPROF            21
#define  SIGIO              22
#define  SIGPOLL            SIGIO
#define  SIGWINCH           23
#define  SIGWINDOW          SIGWINCH
#define  SIGSTOP            24
#define  SIGTSTP            25
#define  SIGCONT            26
#define  SIGTTIN            27
#define  SIGTTOU            28
#define  SIGURG             29
#define  SIGLOST            30

#define  SIG_ERR            ((void (*) ()) (-1))
#define  SIG_DFL            ((void (*) ()) 0)
#define  SIG_IGN            ((void (*) ()) 1)
#define  SIG_HOLD           ((void (*) ()) 3)

#define  SA_ONSTACK         0x00000001
#define  SA_RESETHAND       0x00000004
#define  SA_NOCLDSTOP       0x00000008

#define  SIG_BLOCK          000
#define  SIG_UNBLOCK        001
#define  SIG_SETMASK        002

#define  KILL_ALL_OTHERS    ((pid_t) 0x7fff)

#define  SV_ONSTACK         SA_ONSTACK
#define  SV_BSDSIG          0x00000002
#define  SV_RESETHAND       SA_RESETHAND
```

Figure B-26: <signal.h>

```
struct fp_dbl_block {
   double ss_fp0;        /* Must be double word aligned */
   double ss_fp1;
   double ss_fp2;
   double ss_fp3;
   double ss_fp4;
   double ss_fp5;
   double ss_fp6;
   double ss_fp7;
   double ss_fp8;
   double ss_fp9;
   double ss_fp10;
   double ss_fp11;
   double ss_fp12;
   double ss_fp13;
   double ss_fp14;
   double ss_fp15;
   double ss_fp16;
   double ss_fp17;
   double ss_fp18;
   double ss_fp19;
   double ss_fp20;
   double ss_fp21;
   double ss_fp22;
   double ss_fp23;
   double ss_fp24;
   double ss_fp25;
   double ss_fp26;
   double ss_fp27;
   double ss_fp28;
   double ss_fp29;
   double ss_fp30;
   double ss_fp31;
};
```

Figure B-26: <signal.h>

```
struct fp_int_block {
    int ss_fpstat;          /* Must be double word aligned */
    int ss_fpexcept1;
    int ss_fpexcept2;
    int ss_fpexcept3;
    int ss_fpexcept4;
    int ss_fpexcept5;
    int ss_fpexcept6;
    int ss_fpexcept7;
    int ss_fp4_hi;
    int ss_fp4_lo;
    int ss_fp5_hi;
    int ss_fp5_lo;
    int ss_fp6_hi;
    int ss_fp6_lo;
    int ss_fp7_hi;
    int ss_fp7_lo;
    int ss_fp8_hi;
    int ss_fp8_lo;
    int ss_fp9_hi;
    int ss_fp9_lo;
    int ss_fp10_hi;
    int ss_fp10_lo;
    int ss_fp11_hi;
    int ss_fp11_lo;
    int ss_fp12_hi;
    int ss_fp12_lo;
    int ss_fp13_hi;
    int ss_fp13_lo;
    int ss_fp14_hi;
    int ss_fp14_lo;
    int ss_fp15_hi;
    int ss_fp15_lo;
    int ss_fp16_hi;
    int ss_fp16_lo;
    int ss_fp17_hi;
    int ss_fp17_lo;
    int ss_fp18_hi;
    int ss_fp18_lo;
```

Figure B-26: <signal.h>

```
    int ss_fp19_hi;
    int ss_fp19_lo;
    int ss_fp20_hi;
    int ss_fp20_lo;
    int ss_fp21_hi;
    int ss_fp21_lo;
    int ss_fp22_hi;
    int ss_fp22_lo;
    int ss_fp23_hi;
    int ss_fp23_lo;
    int ss_fp24_hi;
    int ss_fp24_lo;
    int ss_fp25_hi;
    int ss_fp25_lo;
    int ss_fp26_hi;
    int ss_fp26_lo;
    int ss_fp27_hi;
    int ss_fp27_lo;
    int ss_fp28_hi;
    int ss_fp28_lo;
    int ss_fp29_hi;
    int ss_fp29_lo;
    int ss_fp30_hi;
    int ss_fp30_lo;
    int ss_fp31_hi;
    int ss_fp31_lo;
};

struct save_state {
    int ss_flags;                       /* Save State Flags */
    int ss_gr1;                         /* General Registers */
    int ss_rp;
    int ss_gr3;
    int ss_gr4;
    int ss_gr5;
    int ss_gr6;
    int ss_gr7;
    int ss_gr8;
    int ss_gr9;
    int ss_gr10;
```

Figure B-26: <signal.h>

```
    int  ss_gr11;
    int  ss_gr12;
    int  ss_gr13;
    int  ss_gr14;
    int  ss_gr15;
    int  ss_gr16;
    int  ss_gr17;
    int  ss_gr18;
    int  ss_gr19;
    int  ss_gr20;
    int  ss_gr21;
    int  ss_gr22;
    int  ss_arg3;
    int  ss_arg2;
    int  ss_arg1;
    int  ss_arg0;
    unsigned ss_dp;
    unsigned ss_ret0;
    unsigned ss_ret1;
    unsigned ss_sp;
    unsigned ss_gr31;
    unsigned ss_cr11;
    unsigned ss_pcoq_head;
    unsigned ss_pcsq_head;
    unsigned ss_pcoq_tail;
    unsigned ss_pcsq_tail;
    unsigned ss_cr15;
    unsigned ss_cr19;
    unsigned ss_cr20;
    unsigned ss_cr21;
    unsigned ss_cr22;
    unsigned ss_cpustate;
    unsigned ss_sr4;
    unsigned ss_sr0;
    unsigned ss_sr1;
    unsigned ss_sr2;
    unsigned ss_sr3;
    unsigned ss_sr5;
    unsigned ss_sr6;
    unsigned ss_sr7;
```

Figure B-26: <signal.h>

```
    unsigned ss_cr0;
    unsigned ss_cr8;
    unsigned ss_cr9;
    unsigned ss_cr10;
    unsigned ss_cr12;
    unsigned ss_cr13;
    unsigned ss_cr24;
    unsigned ss_cr25;
    unsigned ss_cr26;
    unsigned ss_mpsfu_high;
    unsigned ss_mpsfu_low;
    unsigned ss_mpsfu_ovflo;
    int ss_pad;

    union {
       struct fp_dbl_block  fpdbl;
       struct fp_int_block  fpint;
    }ss_fpblock;

    unsigned ss_cr16;
    unsigned ss_cr23;
};


#define  SS_INTRAP              0x01
#define  SS_INSYSCALL           0x02
#define  SS_ININT               0x04
#define  SS_PSPKERNEL           0x08
#define  SS_ARGSVALID           0x10
#define  SS_DORFI               0x20

#define  FP_TBIT                0x40
```

Figure B-26: <signal.h>

```
struct frame_marker {
    int  fm_edp;
    int  fm_esr4;
    int  fm_erp;
    int  fm_crp;
    int  fm_sl;
    int  fm_clup;
    int  fm_ep;
    int  fm_psp;
};

struct siglocal {
    int  sl_syscall;
    int  sl_onstack;
    int  sl_mask;
    char  sl_syscall_action;
    char  sl_eosys;
    unsigned short  sl_error;
    int  sl_rval1;
    int  sl_rval2;
    int  sl_arg[4];
    struct save_state  sl_ss;  /* See <machine/save_state.h> */
};

struct sigcontext {
    struct siglocal sc_sl;
    int  sc_args[4];
    struct frame_marker  sc_sfm;  /* See <machine/frame.h> */
};

#define  SIG_RETURN        1
#define  SIG_RESTART       0
```

Figure B-26: <signal.h>

```
#define __WORD_MASK      0xFFFFFFFC
#define __DW_MASK        0xFFFFFFF8

typedef double *va_list;

#define va_start(__list,__parmN)                                  \
        __builtin_va_start (__list, &__parmN)

#define va_arg(__list,__mode)                                     \
  (sizeof(__mode) > 8 ?                                           \
    ((__list = (va_list) ((char *)__list - sizeof (int))),        \
     (*((__mode *) (*((int *) (__list)))))) :                     \
    ((__list =                                                    \
     (va_list) ((long)((char *)__list - sizeof (__mode))          \
     & (sizeof(__mode) > 4 ? __DW_MASK : __WORD_MASK))),          \
    (*((__mode *) ((char *)__list +                               \
        ((8 - sizeof(__mode)) % 4))))))

#define va_end(__list)
```

Figure B-27: <stdarg.h>

```
#define  NULL                   0

#define  offsetof(__s_name, __m_name)                  \
      ((size_t)&(((__s_name*)0)->__m_name))

typedef unsigned int          size_t;
typedef unsigned int          wchar_t;
typedef int                   ptrdiff_t;
```

Figure B-28: <stddef.h>

```
#define  _NFILE            60
#define  BUFSIZ            1024

typedef struct {
    int                __cnt;
    unsigned char     *__ptr;
    unsigned char     *__base;
    unsigned short     __flag;
    unsigned char      __fileL;
    unsigned char      __fileH;
} FILE;

#define  _IOFBF            0000000
#define  _IONBF            0000004
#define  _IOEOF            0000020
#define  _IOERR            0000040
#define  _IOLBF            0000200

#define  EOF               (-1)
#define  NULL              0

#define  SEEK_SET          0
#define  SEEK_CUR          1
#define  SEEK_END          2

#define  P_tmpdir          "/var/tmp/"
#define  L_tmpnam          (sizeof(P_tmpdir) + 15)

#define  TMP_MAX           17576
#define  FILENAME_MAX      14
#define  FOPEN_MAX         _NFILE

extern   FILE              __iob[];
extern   int               __flsbuf(unsigned char, FILE *);
extern   int               __filbuf(FILE *);
typedef  long int          fpos_t;

#define  stdin             (&__iob[0])
#define  stdout            (&__iob[1])
#define  stderr            (&__iob[2])
```

Figure B-29: <stdio.h>

```
typedef unsigned int size_t;
typedef double *va_list;

#define  L_ctermid          9
#define  L_cuserid          9

#define  clearerr(__p)                                    \
         ((void) ((__p)->__flag &= ~(_IOERR | _IOEOF)))
#define  feof(__p)          ((__p)->__flag & _IOEOF)
#define  ferror(__p)        ((__p)->__flag & _IOERR)

#define  putc(__c, __p)     (--(__p)->__cnt >= 0 ?        \
         (int) (*(__p)->__ptr++ = (unsigned char) (__c)) : \
         __flsbuf((unsigned char) (__c), (__p)))
#define  getc(__p)          (--(__p)->__cnt >= 0 ?        \
         (int) *(__p)->__ptr++ : __filbuf(__p))

extern char  *optarg;
extern int    opterr;
extern int    optind;
extern int    optopt;
```

Figure B-29: <stdio.h>

```
#define  NULL                   0

#define  EXIT_FAILURE           1
#define  EXIT_SUCCESS           0

#define  MB_CUR_MAX             __nl_char_size
extern int __nl_char_size;

#define  RAND_MAX               32767

typedef unsigned int  size_t;
typedef unsigned int  wchar_t;

typedef struct {
   int  quot;
   int  rem;
} div_t;

typedef struct {
   long int  quot;
   long int  rem;
} ldiv_t;

#define  WIFEXITED(_X)          (((int)(_X)&0377)==0)
#define  WIFSTOPPED(_X)         (((int)(_X)&0377)==0177)
#define  WIFSIGNALED(_X)
         ((((int)(_X)&0377)!=0)&&(((_X)&0377)!=0177))
#define  WEXITSTATUS(_X)        (((int)(_X)>>8)&0377)
#define  WTERMSIG(_X)           ((int)(_X)&0177)
#define  WSTOPSIG(_X)           (((int)(_X)>>8)&0377)
```

Figure B-30: <stdlib.h>

```
#define  NULL  0
typedef unsigned int  size_t;
```

Figure B-31: <string.h>

```
#define CORE_NONE      0x00000000 /* reserved for future use */
#define CORE_FORMAT    0x00000001 /* core version */
#define CORE_KERNEL    0x00000002 /* kernel version */
#define CORE_PROC      0x00000004 /* per process information */
#define CORE_TEXT      0x00000008 /* reserved for future use */
#define CORE_DATA      0x00000010 /* data of the process */
#define CORE_STACK     0x00000020 /* stack of the process */
#define CORE_SHM       0x00000040 /* reserved for future use */
#define CORE_MMF       0x00000080 /* reserved for future use */
#define CORE_EXEC      0x00000100 /* exec information */
#define CORE_FORMAT_VERSION    1

struct corehead {
   int       type;
   space_t   space;
   caddr_t   addr;
   size_t    len;
};

struct proc_exec {
   struct{
       int u_magic;
       struct som_exec_auxhdr som_aux;
   } exdata;
   char cmd[15];
};

struct proc_info {
   int sig;
   int trap_type;
   struct save_state hw_regs;
};
```

Figure 7-32: <sys/core.h>

```
#define  IOCSIZE_MASK        0x1fff0000
#define  IOCPARM_MASK        (IOCSIZE_MASK>>16)
#define  IOC_VOID            0x20000000
#define  IOC_OUT             0x40000000
#define  IOC_IN              0x80000000
#define  IOC_INOUT           (IOC_IN|IOC_OUT)

#define  _IO(x,y)            (IOC_VOID|((x)<<8)|y)
#define  _IOR(x,y,t)                                        \
         (IOC_OUT|((sizeof(t)&IOCPARM_MASK)<<16)|((x)<<8)|y)
#define  _IOW(x,y,t)                                        \
         (IOC_IN|((sizeof(t)&IOCPARM_MASK)<<16)|((x)<<8)|y)
#define  _IOWR(x,y,t)                                       \
      (IOC_INOUT|((sizeof(t)&IOCPARM_MASK)<<16)|((x)<<8)|y)

#define  FIONREAD            _IOR('f', 127, int)
#define  FIONBIO             _IOW('f', 126, int)
#define  FIOASYNC            _IOW('f', 125, int)
#define  FIOSETOWN           _IOW('f', 124, int)
#define  FIOGETOWN           _IOR('f', 123, int)
#define  FIOSNBIO            _IOW('f', 118, int)
#define  FIOGNBIO            _IOR('f', 117, int)

#define  SIOCATMARK          _IOR('s', 7, int)
#define  SIOCSPGRP           _IOW('s', 8, int)
#define  SIOCGPGRP           _IOR('s', 9, int)

#define  SIOCGIFADDR         _IOWR('i',13, struct ifreq)
#define  SIOCGIFDSTADDR      _IOWR('i',15, struct ifreq)
#define  SIOCGIFFLAGS        _IOWR('i',17, struct ifreq)
#define  SIOCGIFBRDADDR      _IOWR('i',18, struct ifreq)
#define  SIOCGIFCONF         _IOWR('i',20, struct ifconf)
#define  SIOCGIFNETMASK      _IOWR('i',21, struct ifreq)
#define  SIOCGIFMETRIC       _IOWR('i',23, struct ifreq)
#define  SIOCSIFMETRIC       _IOW('i',24, struct ifreq)
```

Figure B-33: <sys/ioctl.h>

```
struct ipc_perm {
   uid_t    uid;
   gid_t    gid;
   uid_t    cuid;
   gid_t    cgid;
   mode_t   mode;
   unsigned short  seq;
   key_t    key;
   unsigned short  __ndx;
   unsigned short  __wait;
};

#define  IPC_CREAT            0001000
#define  IPC_EXCL             0002000
#define  IPC_NOWAIT           0004000
#define  IPC_PRIVATE          (key_t)0
#define  IPC_RMID             0
#define  IPC_SET              1
#define  IPC_STAT             2
```

Figure B-34: <sys/ipc.h>

```
#define  PROT_NONE         0x0
#define  PROT_READ         0x1
#define  PROT_WRITE        0x2
#define  PROT_EXEC         0x4

#define  MAP_SHARED        0x01
#define  MAP_PRIVATE       0x02
#define  MAP_FIXED         0x04
#define  MAP_VARIABLE      0x08
#define  MAP_ANONYMOUS     0x10
#define  MAP_FILE          0x20

#define  MADV_NORMAL       0
#define  MADV_RANDOM       1
#define  MADV_SEQUENTIAL   2
#define  MADV_WILLNEED     3
#define  MADV_DONTNEED     4
#define  MADV_SPACEAVAIL   5

#define  MS_SYNC           0x01
#define  MS_ASYNC          0x02
#define  MS_INVALIDATE     0x04

#define  MSEM_UNLOCKED     0x00
#define  MSEM_LOCKED       0x01
#define  MSEM_IF_NOWAIT    0x01
#define  MSEM_IF_WAITERS   0x01

   /* msemaphore must be 16 byte aligned. */

typedef void *msemaphore;
```

Figure B-35: <sys/mman.h>

```
#define  MSG_NOERROR           010000

typedef unsigned short int    msgqnum_t;
typedef unsigned short int    msglen_t;

struct msqid_ds {
   struct ipc_perm  msg_perm;
   struct __msg  *msg_first;
   struct __msg  *msg_last;
   msgqnum_t   msg_qnum;
   msglen_t  msg_qbytes;
   pid_t  msg_lspid;
   pid_t  msg_lrpid;
   time_t  msg_stime;
   time_t  msg_rtime;
   time_t  msg_ctime;
   msglen_t  msg_cbytes;
   char  msg_pad[22];
};
```

Figure B-36: <sys/msg.h>

```
#define PT_SETTRC    0    /* Set Trace */
#define PT_RIUSER    1    /* Read User I Space */
#define PT_RDUSER    2    /* Read User D Space */
#define PT_RUAREA    3    /* Read User Area */
#define PT_WIUSER    4    /* Write User I Space */
#define PT_WDUSER    5    /* Write User D Space */
#define PT_WUAREA    6    /* Write User Area */
#define PT_CONTIN    7    /* Continue */
#define PT_EXIT      8    /* Exit */
#define PT_SINGLE    9    /* Single Step */
#define PT_RUREGS    10   /* Read User Registers */
#define PT_WUREGS    11   /* Write User Registers */
#define PT_ATTACH    12   /* Attach To Process */
#define PT_DETACH    13   /* Detach From Attached Process */
#define PT_RDTEXT    14   /* Read User I Space */
#define PT_RDDATA    15   /* Read User D Space */
#define PT_WRTEXT    16   /* Write User I Space */
#define PT_WRDATA    17   /* Write User D Space */
```

Figure B-37: <sys/ptrace.h>

```
#define  SEM_UNDO            010000

#define  GETNCNT             3
#define  GETPID              4
#define  GETVAL              5
#define  GETALL              6
#define  GETZCNT             7
#define  SETVAL              8
#define  SETALL              9

struct __sem {
   unsigned short int  semval;
   unsigned short int  sempid;
   unsigned short int  semncnt;
   unsigned short int  semzcnt;
 };

struct semid_ds {
   struct ipc_perm  sem_perm;
   struct __sem  *sem_base;
   time_t  sem_otime;
   time_t  sem_ctime;
   unsigned short int  sem_nsems;
   char  sem_pad[22];
};

struct sembuf {
   unsigned short int  sem_num;
   short  sem_op;
   short  sem_flg;
};
```

Figure B-38: <sys/sem.h>

```
#define  SHMLBA              4096
#define  SHM_RDONLY          010000
#define  SHM_RND             020000

typedef unsigned short int shmatt_t;

struct shmid_ds {
   struct ipc_perm  shm_perm;
   int  shm_segsz;
   struct __vas  *shm_vas;
   pid_t  shm_lpid;
   pid_t  shm_cpid;
   shmatt_t  shm_nattch;
   shmatt_t  shm_cnattch;
   time_t  shm_atime;
   time_t  shm_dtime;
   time_t  shm_ctime;
   char  shm_pad[24];
};

#define  SHM_LOCK            3
#define  SHM_UNLOCK          4
```

Figure B-39: <sys/shm.h>

```
struct stat {
   dev_t    st_dev;
   ino_t    st_ino;
   mode_t   st_mode;
   nlink_t  st_nlink;
   unsigned short  st_reserved1;
   unsigned short  st_reserved2;
   dev_t    st_rdev;
   off_t    st_size;
   time_t   st_atime;
   int      st_spare1;
   time_t   st_mtime;
   int      st_spare2;
   time_t   st_ctime;
   int      st_spare3;
   long     st_blksize;
   long     st_blocks;
   unsigned int  st_pad:30;
   unsigned int  st_acl:1;
   unsigned int  st_remote:1;
   dev_t    st_netdev;
   ino_t    st_netino;
   unsigned short  st_reserved3;
   unsigned short  st_reserved4;
   unsigned short  st_reserved5;
   short    st_fstype;
   dev_t    st_realdev;
   unsigned short  st_basemode;
   unsigned short  st_spareshort;
   uid_t    st_uid;
   gid_t    st_gid;
   long     st_spare4[3];
};

#define  S_IFMT         0170000
#define  S_IFREG        0100000
#define  S_IFBLK        0060000
#define  S_IFCHR        0020000
#define  S_IFDIR        0040000
```

Figure B-40: <sys/stat.h>

```
#define  S_IFIFO           0010000
#define  S_IFLNK           0120000

#define  S_ISDIR(_M)       ((_M & S_IFMT)==S_IFDIR)
#define  S_ISCHR(_M)       ((_M & S_IFMT)==S_IFCHR)
#define  S_ISBLK(_M)       ((_M & S_IFMT)==S_IFBLK)
#define  S_ISREG(_M)       ((_M & S_IFMT)==S_IFREG)
#define  S_ISFIFO(_M)      ((_M & S_IFMT)==S_IFIFO)
#define  S_ISLNK(_M)       ((_M & S_IFMT)==S_IFLNK)

#define  S_ENFMT           0002000
#define  S_IFNWK           0110000
#define  S_IFSOCK          0140000
#define  S_ISVTX           0001000

#define  S_ISNWK(_M)       ((_M & S_IFMT)==S_IFNWK)
#define  S_ISSOCK(_M)      ((_M & S_IFMT)==S_IFSOCK)

#define  S_ISGID           0002000
#define  S_ISUID           0004000

#define  S_IRWXU           0000700
#define  S_IRUSR           0000400
#define  S_IWUSR           0000200
#define  S_IXUSR           0000100

#define  S_IRWXG           0000070
#define  S_IRGRP           0000040
#define  S_IWGRP           0000020
#define  S_IXGRP           0000010

#define  S_IRWXO           0000007
#define  S_IROTH           0000004
#define  S_IWOTH           0000002
#define  S_IXOTH           0000001
```

Figure B-40: <sys/stat.h>

```
#define  DELIVERY_SIGNALS      1
#define  TIMEOFDAY             1

typedef long timer_t;

struct timespec {
   unsigned long tv_sec;
   long tv_nsec;
};

struct itimerspec {
   struct timespec it_interval;
   struct timespec it_value;
};
```

Figure 7-41: <sys/timers.h>

```
struct tms {
    clock_t tms_utime;
    clock_t tms_stime;
    clock_t tms_cutime;
    clock_t tms_cstime;
};
```

Figure B-42: <sys/times.h>

```
typedef long                    dev_t;
typedef unsigned long           ino_t;
typedef unsigned short          mode_t;
typedef short                   nlink_t;
typedef long                    off_t;
typedef long                    pid_t;
typedef long                    gid_t;
typedef long                    uid_t;
typedef long                    time_t;
typedef unsigned int            size_t;
typedef int                     ssize_t;
typedef unsigned long           clock_t;
typedef long                    key_t;
typedef long                    daddr_t;
typedef char                    *caddr_t;
typedef long                    swblk_t;
typedef unsigned char           ubit8;
typedef unsigned short          ubit16;
typedef unsigned long           ubit32;
typedef char                    sbit8;
typedef short                   sbit16;
typedef long                    sbit32;

#define  UID_NO_CHANGE          ((uid_t) -1)
#define  GID_NO_CHANGE          ((gid_t) -1)

typedef long                    fd_mask;

#define  howmany(x,y)           (((x)+((y)-1))/(y))

typedef struct fd_set {
   fd_mask  fds_bits[howmany(2048, (sizeof(fd_mask) * 8))];
} fd_set;

#define  FD_SET(n,p)                                        \
    ((p)->fds_bits[(n)/(sizeof(fd_mask) * 8)] |=            \
         (1 << ((n) % (sizeof(fd_mask) * 8))))

#define  FD_ZERO(p)                                         \
         memset((char *)(p), (char) 0, sizeof(*(p)))
```

Figure B-43: <sys/types.h>

```
#define  FD_CLR(n,p)                                      \
   ((p)->fds_bits[(n)/(sizeof(fd_mask) * 8)] &=           \
        ~(1 << ((n) % (sizeof(fd_mask) * 8))))
#define  FD_ISSET(n,p)                                    \
   ((p)->fds_bits[(n)/(sizeof(fd_mask) * 8)] &            \
        (1 << ((n) % (sizeof(fd_mask) * 8))))
```

Figure B-43: <sys/types.h>

```
struct utsname {
   char  sysname[9];
   char  nodename[9];
   char  release[9];
   char  version[9];
   char  machine[9];
   char  __idnumber[15];
};
```

Figure B-44: <sys/utsname.h>

```
typedef long   fsid_t[2];

struct statfs {
   long  f_type;
   long  f_bsize;
   long  f_blocks;
   long  f_bfree;
   long  f_bavail;
   long  f_files;
   long  f_ffree;
   fsid_t  f_fsid;
   long  f_magic;
   long  f_featurebits;
   long  f_spare[4];
   site_t  f_cnode;
   short  f_pad;
};

#define  MAXFIDSZ            16
```

Figure B-45: <sys/vfs.h>

```
#define  WNOHANG             1
#define  WUNTRACED           2

#define  WIFEXITED(_X)       (((int)(_X)&0377)==0)
#define  WIFSTOPPED(_X)      (((int)(_X)&0377)==0177)
#define  WIFSIGNALED(_X)
         ((((int)(_X)&0377)!=0)&&(((_X)&0377)!=0177))
#define  WEXITSTATUS(_X)     (((int)(_X)>>8)&0377)
#define  WTERMSIG(_X)        ((int)(_X)&0177)
#define  WSTOPSIG(_X)        (((int)(_X)>>8)&0377)
```

Figure B-46: <sys/wait.h>

```
#define  NCCS           32

typedef unsigned int    tcflag_t;
typedef unsigned char   cc_t;
typedef unsigned int    speed_t;

struct termios {
   tcflag_t  c_iflag;
   tcflag_t  c_oflag;
   tcflag_t  c_cflag;
   tcflag_t  c_lflag;
   tcflag_t  c_reserved;
   cc_t  c_cc[NCCS];
};

#define  IGNBRK         0000001
#define  BRKINT         0000002
#define  IGNPAR         0000004
#define  PARMRK         0000010
#define  INPCK          0000020
#define  ISTRIP         0000040
#define  INLCR          0000100
#define  IGNCR          0000200
#define  ICRNL          0000400
#define  IUCLC          0001000
#define  IXON           0002000
#define  IXANY          0004000
#define  IXOFF          0010000
#define  IMAXBEL        0040000

#define  OPOST          0000001
#define  OLCUC          0000002
#define  ONLCR          0000004
#define  OCRNL          0000010
#define  ONOCR          0000020
#define  ONLRET         0000040
#define  OFILL          0000100
#define  OFDEL          0000200
#define  NLDLY          0000400
```

Figure B-47: <termios.h>

```
#define  NL0        0
#define  NL1        0000400
#define  CRDLY      0003000
#define  CR0        0
#define  CR1        0001000
#define  CR2        0002000
#define  CR3        0003000
#define  TABDLY     0014000
#define  TAB0       0
#define  TAB1       0004000
#define  TAB2       0010000
#define  TAB3       0014000
#define  BSDLY      0020000
#define  BS0        0
#define  BS1        0020000
#define  VTDLY      0040000
#define  VT0        0
#define  VT1        0040000
#define  FFDLY      0100000
#define  FF0        0
#define  FF1        0100000
#define  XTABS      TAB3

#define  EXTA       0000036
#define  EXTB       0000037
#define  CLOCAL     0010000
#define  CREAD      0000400
#define  CSIZE      0000140
#define  CS5        0
#define  CS6        0000040
#define  CS7        0000100
#define  CS8        0000140
#define  CSTOPB     0000200
#define  HUPCL      0004000
#define  PARENB     0001000
#define  PARODD     0002000
#define  LOBLK      0020000

#define  ISIG       0000001
#define  ICANON     0000002
```

Figure B-47: <termios.h>

```
#define   XCASE          0000004

#define   ECHO           0000010
#define   ECHOE          0000020
#define   ECHOK          0000040
#define   ECHONL         0000100
#define   NOFLSH         0000200
#define   ECHOCTL        0000400
#define   ECHOPRT        0001000
#define   ECHOKE         0002000
#define   FLUSHO         0004000
#define   PENDIN         0010000

#define   TOSTOP         0020000
#define   IEXTEN         0040000

#define   VINTR          0
#define   VQUIT          1
#define   VERASE         2
#define   VKILL          3
#define   VEOF           4
#define   VEOL           5
#define   VEOL2          6

#define   VMIN           11
#define   VTIME          12
#define   VSUSP          13
#define   VSTART         14
#define   VSTOP          15
#define   VREPRINT       16
#define   VDISCARD       17
#define   VWERASE        18
#define   VLNEXT         19
#define   VDSUSP         20

#define   CBAUD          0000037
#define   B0             0
#define   B50            0000001
#define   B75            0000002
#define   B110           0000003
```

Figure B-47: <termios.h>

```
#define  B134           0000004
#define  B150           0000005
#define  B200           0000006
#define  B300           0000007
#define  B600           0000010
#define  B900           0000011
#define  B1200          0000012
#define  B1800          0000013
#define  B2400          0000014
#define  B3600          0000015
#define  B4800          0000016
#define  B7200          0000017
#define  B9600          0000020
#define  B19200         0000021
#define  B38400         0000022
#define  B57600         0000023
#define  B115200        0000024
#define  B230400        0000025
#define  B460800        0000026

#define  TCSANOW        0
#define  TCSADRAIN      1
#define  TCSAFLUSH      2
#define  TCIFLUSH       0
#define  TCOFLUSH       1
#define  TCIOFLUSH      2
#define  TCOOFF         0
#define  TCOON          1
#define  TCIOFF         2
#define  TCION          3
#define  SSPEED         7

struct winsize {
   unsigned short  ws_row;
   unsigned short  ws_col;
   unsigned short  ws_xpixel;
   unsigned short  ws_ypixel;
};
#define  TIOCGWINSZ            _IOR('t', 107, struct winsize)
#define  TIOCSWINSZ            _IOW('t', 106, struct winsize)
```

Figure B-47: <termios.h>

```
#define  NULL                  0
#define  CLOCKS_PER_SEC        1000000

typedef unsigned long          clock_t;
typedef long                   time_t;
typedef unsigned int           size_t;

extern long                    timezone;
extern int                     daylight;
extern char                    *tzname[2];

struct tm {
   int  tm_sec;
   int  tm_min;
   int  tm_hour;
   int  tm_mday;
   int  tm_mon;
   int  tm_year;
   int  tm_wday;
   int  tm_yday;
   int  tm_isdst;
};

struct timeval {
    unsigned long tv_sec;
    long tv_usec;
};

struct itimerval {
   struct timeval  it_interval;
   struct timeval  it_value;
};

#define  CLK_TCK               100

#define  ITIMER_REAL           0
#define  ITIMER_VIRTUAL        1
#define  ITIMER_PROF           2
```

Figure B-48: <time.h>

```
#define  UL_GETFSIZE         1
#define  UL_SETFSIZE         2
#define  UL_GETMAXBRK        3
```

Figure B-49: <ulimit.h>

```
/* Values represented by an asterisk are dynamic and
   should be determined using sysconf() or pathconf()

#define  _POSIX_CHOWN_RESTRICTED *
#define  _POSIX_NO_TRUNC         *
#define  _XOPEN_ENH_I18N         *
#define  _POSIX2_LOCALEDEF       *
*/
#define  _POSIX_SAVED_IDS        1
#define  _POSIX_JOB_CONTROL      2
#define  _POSIX_VDISABLE         0xff

#define  R_OK                    4
#define  W_OK                    2
#define  X_OK                    1
#define  F_OK                    0

#define  SEEK_SET                0
#define  SEEK_CUR                1
#define  SEEK_END                2

#define  STDIN_FILENO            0
#define  STDOUT_FILENO           1
#define  STDERR_FILENO           2

#define  _SC_ARG_MAX             0
#define  _SC_CHILD_MAX           1
#define  _SC_CLK_TCK             2
#define  _SC_NGROUPS_MAX         3
#define  _SC_OPEN_MAX            4
#define  _SC_JOB_CONTROL         5
```

Figure B-50: <unistd.h>

```
#define  _SC_SAVED_IDS          6
#define  _SC_PASS_MAX           9
#define  _SC_STREAM_MAX         100
#define  _SC_TZNAME_MAX         101
#define  _SC_1_VERSION          102
#define  _SC_VERSION            _SC_1_VERSION
#define  _SC_BC_BASE_MAX        200
#define  _SC_BC_SCALE_MAX       202
#define  _SC_EXPR_NEST_MAX      204
#define  _SC_LINE_MAX           205
#define  _SC_RE_DUP_MAX         207
#define  _SC_2_VERSION          211
#define  _SC_2_C_BIND           212
#define  _SC_2_C_DEV            213
#define  _SC_2_FORT_DEV         214
#define  _SC_2_SW_DEV           215
#define  _SC_2_C_VERSION        216
#define  _SC_2_CHAR_TERM        217
#define  _SC_2_FORT_RUN         218
#define  _SC_2_LOCALEDEF        219
#define  _SC_2_UPE              220
#define  _SC_BC_STRING_MAX      221
#define  _SC_COLL_WEIGHTS_MAX   222
#define  _SC_CLOCKS_PER_SEC     2000
#define  _SC_XOPEN_VERSION      2001
#define  _SC_XOPEN_CRYPT        2002
#define  _SC_XOPEN_ENH_I18N     2003
#define  _SC_XOPEN_SHM          2004
#define  _SC_ABS_OS_VERSION     3000
#define  _SC_PAGE_SIZE          3001
#define  _SC_ATEXIT_MAX         3002
#define  _SC_SECURITY_CLASS     10000
#define  _SC_CPU_VERSION        10001
#define  _SC_IO_TYPE            10002
#define  _SC_MSEM_LOCKID        10003
#define  _SC_MCAS_OFFSET        10004

#define  _PC_LINK_MAX           0
#define  _PC_MAX_CANON          1
```

Figure B-50: <unistd.h>

```
#define  _PC_MAX_INPUT          2
#define  _PC_NAME_MAX           3
#define  _PC_PATH_MAX           4
#define  _PC_PIPE_BUF           5
#define  _PC_CHOWN_RESTRICTED   6
#define  _PC_NO_TRUNC           7
#define  _PC_VDISABLE           8
#define  _XOPEN_VERSION         4
#define  _XOPEN_XPG4         1
#define  _XOPEN_XCU_VERSION  4
#define  _XOPEN_CRYPT        1
#define  _XOPEN_SHM          1

#define  _AES_OS_VERSION     1
#define  _POSIX_VERSION      199009L
#define  _POSIX2_VERSION     199209L
#define  _POSIX2_C_VERSION   199209L
#define  _POSIX2_C_BIND      1
#define  _POSIX2_C_DEV       1
#define  _POSIX2_FORT_DEV    1
#define  _POSIX2_FORT_RUN    1
#define  _POSIX2_SW_DEV      1
#define  _POSIX2_CHAR_TERM   1
#define  _POSIX2_UPE         1

#define  _CS_PATH            200

#define  F_ULOCK             0
#define  F_LOCK              1
#define  F_TLOCK             2
#define  F_TEST              3

#define  GF_PATH      "/etc/group"
#define  PF_PATH      "/etc/passwd"
#define  IN_PATH      "/usr/include"
#define  CS_PATH      "/usr/bin:/usr/ccs/bin:"

#define  SEC_CLASS_NONE      0
#define  SEC_CLASS_C2        1
```

Figure B-50: <unistd.h>

```
#define  SEC_CLASS_B1          2
#define  IO_TYPE_WSIO          01
#define  IO_TYPE_SIO           02

#define  CPU_PA_RISC1_0        0x20B
#define  CPU_PA_RISC1_1        0x210
#define  CPU_PA_RISC2_0        0x214
#define  CPU_PA_RISC_MAX       0x2FF

#define  CPU_IS_PA_RISC(__x)   ((__x) == CPU_PA_RISC1_0 || \
                                (__x) == CPU_PA_RISC1_1 || \
                                (__x) == CPU_PA_RISC2_0)

extern char                    *optarg;
extern int                     opterr;
extern int                     optind;
extern int                     optopt;

#define  NULL                  0
```

Figure B-50: <unistd.h>

```
struct utimbuf {
   time_t actime;
   time_t modtime;
};
```

Figure B-51: <utime.h>

```
typedef double *va_list;
int va_alist;

#define  va_start(__list)       ((__list)=(char *)&va_alist+4)

#define  va_arg(__list,__mode)                              \
  (sizeof(__mode) > 8 ?         \
    ((__list = (va_list) ((char *)__list - sizeof (int))),\
     (*((__mode *) (*((int *) (__list)))))) :\
    ((__list =                                              \
      (va_list) ((long)((char *)__list - sizeof (__mode))\
      & (sizeof(__mode) > 4 ? 0xFFFFFFF8 : 0xFFFFFFFC))),\
    (*((__mode *) ((char *)__list +                         \
      ((8 - sizeof(__mode)) % 4))))))

#define va_end(__list)

#define  va_dcl                   long va_alist;
```

Figure B-52: <varargs.h>

```
typedef unsigned int          wint_t;
typedef unsigned int          wchar_t;
typedef unsigned int          wctype_t;

#define  WEOF                 (wint_t)(-1)
```

Figure B-53: <wchar.h>

```
typedef struct {
   size_t  we_wordc;
   char  **we_wordv;
   size_t  we_offs;
} wordexp_t;

#define  WRDE_APPEND           0x01
#define  WRDE_DOOFFS           0x02
#define  WRDE_NOCMD            0x04
#define  WRDE_REUSE            0x08
#define  WRDE_SHOWERR          0x10
#define  WRDE_UNDEF            0x20
#define  WRDE_BADCHAR          1
#define  WRDE_BADVAL           2
#define  WRDE_CMDSUB           3
#define  WRDE_NOSPACE          4
#define  WRDE_SYNTAX           5
#define  WRDE_INTERNAL         6
#define  WRDE_NOSYS            (-1)
```

Figure B-54: <wordexp.h>

C

Index

T

U

V

W