HEWLETT ® PACKARD

64-Bit Runtime Architecture for PA-RISC 2.0

Version 3.3
October 6, 1997

Legal Notices

Table of Contents

1

Introduction

This document describes common software conventions for user-mode applications running in “Wide” mode on the PA-RISC 2.0 architecture. It does not define operating-system interfaces or any conventions specific to any operating environment.

The runtime architecture defines many, but not all, of the conventions necessary to compile, link, and execute a program on an operating system that supports these conventions. Its purpose is to ensure that object modules produced by different compilers can be linked together into a single application, and to specify the interfaces between compilers and the linker. A complete specification of the runtime environment for a given operating system would include this document, supplemented by the following specifications:

- The *PA-RISC 2.0 Instruction Set and Reference Manual*. Only the parts of this reference that describe the user-mode instruction set and machine state are relevant to this specification.
- The *Application Programming Interface* (API) specification for the operating system. This defines the set of system interfaces available to the application.
- A specification of binary values for symbolic constants used by the API. These are generally provided by the system header files provided by the operating system vendor that are used at compile time.
- Environment-specific conventions for the combination of the operating system and the PA-RISC 2.0 architecture. These conventions supplement the general conventions in this document with details on dynamic linking, stack unwinding, traps and exception handling, and process initialization.
- The object file formats used for relocatable objects produced by compilers, and for loadable objects produced by linkers. These are published separately because no one format is supported by all operating systems.

Together, this set of specifications comprise what is known as an *Application Binary Interface*, or ABI. An ABI may deliberately exclude some of the environment-specific conventions, however, so an application that depends on any of the excluded features may not be portable to other platforms that conform to that ABI.

The runtime architecture specification is intended to be environment-neutral, yet detailed enough so compilers have few dependencies on the operating environment: the same compiler can compile an application for any operating environment, given the appropriate set of header files. In practice, however, there are some necessary exceptions to this ideal. Notably, the object file format is not part of this specification, so compilers must be prepared to target multiple object file formats. User threads and exception handling details may also affect the compiler.

2

Memory Model

This chapter describes the aspects of the runtime architecture that are related to the address space of a user-mode process.

PA-RISC 2.0 supports two addressing modes, narrow and wide, controlled by the W bit in the Processor Status Word (PSW). Narrow-mode operation is designed for 32-bit programs, and provides a compatible form of addressing for applications compiled for earlier versions of the PA-RISC architecture. This document applies only to 64-bit programs, which will always run in the processor's wide mode.

In wide mode, the processor provides a global virtual address space of 2^{96} bytes. Each application is provided a subset of this virtual address space consisting of 2^{64} bytes, which is divided into four quadrants. Each quadrant is mapped into a portion of the global virtual address space by means of four space registers, which are entirely under the control of the operating system.

The runtime architecture grants a significant amount of freedom to the operating system with respect to the way a user-mode process uses its virtual address space. Some operating systems, such as HP-UX, use a segmented address space, in which memory is shared by mapping the same global virtual address into each process' address space. Other operating systems may provide each process with its own private, flat, address space, sharing data by means of virtual address aliasing. The difference between these two approaches requires that the runtime architecture leave a number of details open. This chapter defines those details that are common to all environments; a compiler should not assume anything about the address space that is not specified here.

2.1 Sections and Segments

Compilers separate different parts of the generated object code and data into different sections. At link time, as object files are combined, sections with the same name are collected together and consolidated, producing an output file with one instance of each type of section. These output file sections are further grouped into a few segments, which are then treated as contiguous units by the program loader.

The purpose of sections is to allow the compiler to generate separate pieces of code and data that can be combined with other similar pieces from other object files by the linker. This provides a mechanism for achieving locality of reference, and for assuring addressability to the contents of those sections. The most important attribute of a section is the type of access permitted to pages allocated to the section; all contributions to a single section share the same minimum set of access rights.

The purpose of segments is to allow the linker to group the sections into a fewer number of program regions. Each segment has a unique means of addressability, and the sections common to a segment

are all addressed by the same means. A compiler may assume that any two locations within the same segment will have a fixed relationship relative to each other when the program is executing, but may not assume the same for two locations in different segments.

The runtime architecture also defines some additional segments that do not receive any contributions directly from the compiled object files. These segments—heap, stack, and shared memory segments—are allocated either at program load time or dynamically during program execution.

Table 2-1 lists the sections and segments defined by the runtime architecture and their attributes. Each section corresponds directly to a section in the object files generated by the compilers; the names and other properties of these object file sections are specific to the object file format, and are described in a separate document. The section names shown here apply to the ELF object file format.

Table 2-1. Segments and sections

Segment	Addressability	Section	Minimum Access	Purpose
Text	IA-relative	.rodata	R	Literals and read-only constants
		.text	X	Object code
		.unwind	R	Stack unwind table
		.unwind_info	R	Stack unwind info blocks
Data	gp-relative	.sdata	R, W	Short initialized data
		.sbss	R, W	Short uninitialized data
		.dlt	R, W	Linkage tables
		.data	R, W	Long initialized data
		.bss	R, W	Long uninitialized data
		.hbss	R, W	Huge uninitialized data
Thread Data	tp-relative	.tbss	R, W	Thread-specific uninit. data
Stack	sp-relative		R, W	Process/thread stacks
Heap	indirect			Dynamically-allocated data
Shared Data	indirect			Shared memory

A program consists of one or more load modules; each load module corresponds to a separately-linked component of the program. The main program is one load module, and each DLL used by the program is another load module.

Each load module consists of one text segment, one data segment, and one optional thread data segment. An executing process also contains one stack segment per thread, plus any number of heap and shared data segments.

None of the sections in the text segment require write access, and the runtime architecture requires that nothing in the text segment may require load-time relocation. This permits the operating system to share a load module's text segment across multiple processes.

The data segment, including linkage tables, is addressed via the global pointer (gp), which is defined as GR 27 for both main program and DLLs. It must be treated as a scratch register by the compilers, so it must be saved and restored across procedure calls. The gp value is unique to each load module, and is established as part of the calling sequence. This is discussed further in Chapter 5. The .dlt section, containing the linkage tables, is built by the linker; compiled objects should not contribute directly to this section.

The thread data segment is addressed by the thread pointer (tp), which is defined as CR 27. This segment contains all the thread-local storage allocated by the compiled objects. The operating system will allocate a separate thread data segment to each running thread in a process, so each thread will have a unique value for its thread pointer. Compilers may allocate data directly in .tbss sections, but all thread-local storage must be uninitialized (it will be initialized at run time to zeroes). Compilers may also declare thread-local common blocks, which the linker will resolve and allocate if necessary. At program load time, the dynamic loader will concatenate the thread-local storage segments from all load modules and assign final tp-relative offsets to all thread-local data items. References to thread-local data items, therefore, must use a tp-relative offset obtained from the linkage table. Example code sequences may be found in Appendix A.

2.2 PIC Programming Model

Unlike the 32-bit runtime architecture for PA-RISC, the 64-bit runtime architecture defines a single compilation model, PIC (Position-Independent Code).

In the PIC model, all code must be both position independent and "binding independent." The latter term means that the compiler may not assume that external references to data can be resolved at link time, so these references must be made indirectly through a linkage table.

In lieu of the non-PIC model, several new optimization options allow the compilers to generate more efficient code, by providing certain assurances about the program. For example, one option directs the compilers to assume that certain global symbols will be defined in the same load module, so that it can generate more efficient code to access a variable or call a procedure. Some of these optimizations are illustrated in Appendix A.

All code must be position independent in the sense that it has no dependency on its own absolute address or on its position relative to any other program segments (i.e., data segments, or other load modules' text segments). This means that all branching must be pc-relative, and accesses to read-only data in the text segment must be either pc-relative or indirect through the linkage table. (The compiler may choose instead to put some or all read-only data in the data segment.)

2.3 Spaces and Space Registers

The OS will use a different address space layout for 64-bit processes, so we will not be able to specify that main program text is at the low end of Quadrant 0, nor will we be able to use absolute addressing at all. This will affect millicode calls, long calls, plabel materialization, and non-PIC literal references.

Compilers must avoid any expicit reference to space registers, so there is no need to specify any association between particular segments and quadrants of the address space.

2.4 Partitioning Data

The data segment is partitioned into short, long, and huge regions (the .sdata, .data, .sbss, .bss, and .hbss sections). Short "own" data is placed in .sdata and .sbss; "huge" uninitialized data is placed in .hbss; all other data goes in .data and .bss. The linker builds the linkage table in the short data segment.

"Own" data is data that is known, at compile time, to be free from the possibility of pre-emption by another load module. That is, if a data item has external scope, such that a symbolic reference may be overridden by another definition in another load module, any references to that item must be indirect through the linkage table, and should not be placed in the short region.

The size threshold for placing data in the short region is 8 bytes. Any single item whose size is 8 bytes or less may be placed in the short region; all items larger than 8 bytes should be placed in the long region. Because 16 bits of displacement is too small to guarantee one-instruction access to short data, compilers should normally use an ADDIL/LDD sequence for all data; thus, they do not need to use the linkage table to access long "own" data. Under certain circumstances, a compiler may choose to use a "tiny" addressing model for all short data, omitting the ADDIL instruction. The tiny addressing model should be used only when the compiler has a reasonable expectation that the short data region in the load module will be no larger than 64K (2^{16}) bytes (for example, via a compiler switch set by the user or based on information gathered during whole-program optimization).

The "huge" region is designated for large data items that may cause the data segment to grow beyond 2G (2^{31}) bytes. These data items are placed farthest from the global pointer, and the compiler must always use indirect addressing via the linkage table to address them, ensuring that the short and long regions, together, can be addressed with the ADDIL/LDD instruction sequence. The exact criteria for placing a particular data item in the huge region is unspecified here, and is the responsibility of the compiler to determine when to use the huge addressing model. A compiler may set a size threshold (to prevent several data items, each under the threshold, from combining to overflow the long region size limit), or it may support a compiler switch set by the user to force the allocation of certain classes of data in the huge region.

If any data allocated in the huge region has static initialization, the compiler should generate a run-time initialization rather than attempt to allocate the entire data item in the initialized data region.

2.5 Interoperability with 32-Bit Applications

Because the address space of a 32-bit application is limited, an operating system may require a 64-bit application to specify which, if any, shared memory segments are expected to be shared with a 32-bit application. This will allow the operating system to allocate the segment in an area of memory where it can be addressed by both processes.

2.6 Limits and Assumptions

The data segment is guaranteed to be at least 32K bytes (8 pages) from a quadrant boundary (not 4K), so any offset that will fit in the displacement field of a load or store instruction can be used without danger of using the wrong space register in zero-base addressing.

Each text segment is restricted to 2G (2^{31}) bytes in size; this allows the compiler to use ADDIL/LDO sequences to materialize pc-relative offsets for read-only data.

The short and long regions of each data segment, combined (consisting of initialized and uninitialized data), are restricted to 2G (2^{31}) bytes in size. This allows the compiler to use direct gp-relative addressing to reach long own data.

Stack frames are not restricted in size, but any frame of size 1G (2^{30}) bytes or larger must be treated as a dynamic frame for unwind purposes, since the unwind descriptor is not able to represent any size larger than this.

3

Data Representation

Applications running in a 64-bit environment use the "LP64" data model: ints are 32 bits, while longs and pointers are 64 bits. All data is allocated in the Big-Endian byte order.

Within this specification, the term *halfword* refers to a 16-bit object, the term *word* refers to a 32-bit object, the term *doubleword* refers to a 64-bit object, and the term *quadword* refers to a 128-bit object.

The following sections define the size, alignment requirements, and hardware representation of the standard C and Fortran datatypes.

3.1 Fundamental Types

Table 3-1 lists the scalar datatypes supported by the architecture. Sizes and alignments are shown in bytes.

Table 3-1. Scalar Types

Type	C	Size	Alignment	Hardware Representation
Integral	char signed char	1	1	signed byte
	unsigned char	1	1	unsigned byte
	short signed short	2	2	signed halfword
	unsigned short	2	2	unsigned halfword
	int signed int enum	4	4	signed word
	unsigned int	4	4	unsigned word
	long signed long long long signed long long	8	8	signed doubleword
	unsigned long unsigned long long	8	8	unsigned doubleword
Pointer	*any-type* * *any-type* (*) ()	8	8	unsigned doubleword

Table 3-1. Scalar Types

Type	C	Size	Alignment	Hardware Representation
Floating-point	float	4	4	IEEE single precision
	double	8	8	IEEE double precision
	__float80	16	16	IEEE 80-bit precision
	long double	16	16	IEEE 128-bit precision

A null pointer (for all types) has the value zero.

Note: Although the 80-bit floating-point data type is shown in the table, compilers are not required to support it. If supported, however, it must conform to the size and alignment restrictions listed here.

3.2 Aggregate Types

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component. The size of any object, including aggregates and unions, is always a multiple of the object's alignment. An array uses the same alignment as its elements. Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding is undefined.

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.
- A structure's size is increased, if necessary, to make it a multiple of the alignment. This may require *tail padding*, depending on the last member.

In the following examples, members' byte offsets appear in the upper right corners for little-endian, in the upper left for big-endian.

```
struct {
    char   c;
};
```

Byte aligned, sizeof is 1

Figure 3-1. Structure Smaller Than a Word

```
struct {
    char   c;
    char   d;
    short  s;
    int n;
};
```

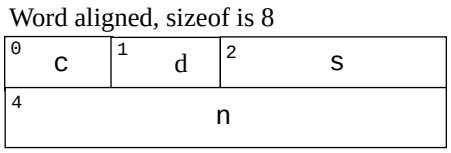

Figure 3-2. No Padding

```
struct {
    char   c;
    short  s;
};
```

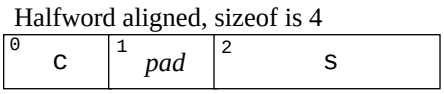

Figure 3-3. Internal Padding

```
struct {
    char   c;
    double d;
    short  s;
 };
```

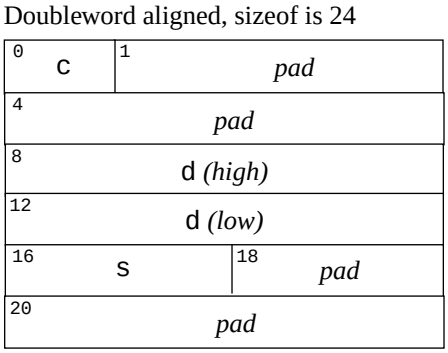

Figure 3-4. Internal and Tail Padding

```
union {
    char   c;
    short  s;
    int j;
 };
```

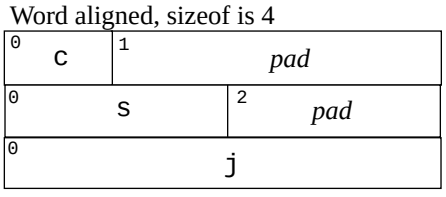

Figure 3-5. union **Allocation**

3.3 Bit Fields

C struct and union definitions may have *bit-fields* that define integral objects with a specified number of bits. Table 3-2 shows the allowable widths of bit fields of each base type, and the corresponding ranges.

Table 3-2. Bit Field Ranges

Base Type	Width *w*	Range
char unsigned char	1 to 8	0 to 2^w-1
signed char	1 to 8	-2^{w-1} to $2^{w-1}-1$
short unsigned short	1 to 16	0 to 2^w-1
signed short	1 to 16	-2^{w-1} to $2^{w-1}-1$
int unsigned int enum	1 to 32	0 to 2^w-1
signed int	1 to 32	-2^{w-1} to $2^{w-1}-1$
long unsigned long	1 to 64	0 to 2^w-1
signed long	1 to 64	-2^{w-1} to $2^{w-1}-1$

Bit fields whose base types (excluding enumerated types) are given without a signed or unsigned qualifier are treated as unsigned.

Bit fields of enumerated types are considered signed unless an unsigned type is necessary to represent all the enumeration constants of the enumeration type.

Bit-fields obey the same size and alignment rules as other structure and union members, with the following additions:

- Bit-fields are allocated from right to left (least to most significant) for little endian. They are allocated left to right (most to least significant) for big-endian.
- A bit-field must entirely reside in a storage unit appropriate for its declared type. For example, a bit field of type short must never cross a halfword boundary.
- Bit-fields may share a storage unit with other struct/union members, including members that are not bit-fields. Of course, each struct member occupies a different part of the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union. Zero-length unnamed bit-fields force the aligment of subsequent members to the boundary corresponding to the size of bit-field.

The following examples show struct and union members' byte offsets in the upper corners; bit numbers appear in the lower corners.

```
0xF1F2F3F4
```

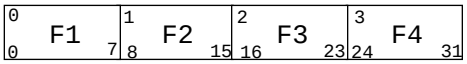

Figure 3-6. Bit Numbering

```
struct {
    int    j:5;
    int    k:6;
    int    m:7;
};
```

Word aligned, sizeof is 4

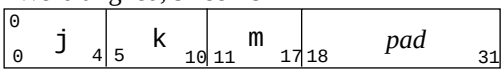

Figure 3-7. Bit Field Allocation

```
struct {
    short  s:9;
    long   j:9;
    char   c;
    short  t:9;
    short  u:9;
    char   d;
};
```

Doubleword aligned, sizeof is 16

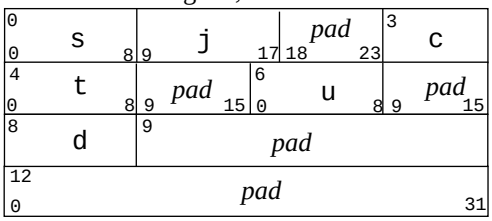

Figure 3-8. Boundary Alignment

```
struct {
    char   c;
    short  s:8;
    };
```

Halfword aligned, sizeof is 2

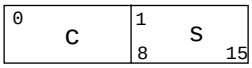

Figure 3-9. Storage Unit Sharing

```
union {
    char   c;
    short  s:8;
    };
```

Halfword aligned, sizeof is 2

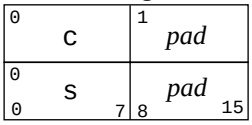

Figure 3-10. union Allocation

```
struct {
    char   c;
    int    :0;
    char   d;
    short  :9;
    char   e;
    char   :0;
    };
```

Byte aligned, sizeof is 9

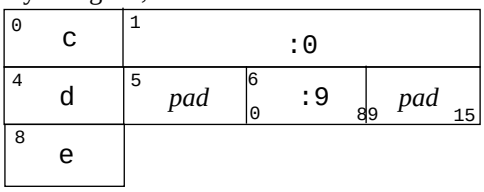

Figure 3-11. Unnamed Bit-Fields

Note that unnamed bit fields do not affect the alignment of the structure.

As the examples show, int and long bit-fields (including signed and unsigned) usually pack more densely than smaller base types. One can use char and short bit-fields to force allocation within those types, but int is generally more efficient.

3.4 Fortran Data Types

Table 3-3 shows the correspondence between ANSI Fortran's scalar types and the processor's data types. ANSI Fortran requires REAL and INTEGER to be the same size. Many Fortran compilers allow INTEGER*n, LOGICAL*n, and REAL*n to specify specific processor sizes. ("n" is in bytes). The COMPLEX datatype is treated exactly the same as a C structure composed of two float members.

Table 3-3. Fortran Datatypes

Type	Fortran	Size	Alignment (bytes)	Hardware Representation
Character	CHARACTER*_n_	_n_	1	byte
Integral	LOGICAL	4	4	doubleword
	INTEGER	4	4	signed doubleword
Floating-point	REAL	4	4	IEEE single-precision
	DOUBLE PRECISION	8	8	IEEE double-precision
	COMPLEX	8	4	2 IEEE single-precision

4

Register Usage

This chapter describes the conventions for using processor registers in an application. There are two areas of concern to a user-mode application: what registers are available, and the conventions for using registers across standard procedure calls.

The former is limited both by the architecture, which defines what registers may be accessed by user-mode programs, and by the operating system, which determines what registers to save when a process is context-switched.

The conventions for using registers across standard procedure calls place most registers into one of two categories:

- Scratch registers, whose values may not live across a procedure call; and
- Preserved registers, whose values are preserved across a procedure call.

If a procedure has a value in a scratch register that is needs to remain live across a procedure call, it must save the value in the stack (or in a preserved register) prior to making the procedure call. Scratch registers are also called caller-saves registers.

Before a procedure can use a preserved register for its own purposes, it must save that register, then restore it prior to returning. Preserved registers are also called callee-saves registers.

Some registers are not so easily placed into one of these two categories, and are considered special purpose.

4.1 General Registers

The PA-RISC 2.0 architecture provides 32 general registers, each 64 bits wide. All 32 registers may be used by a user-mode application. The runtime architecture defines the following conventions for using these registers across standard procedure calls:

- GR 0 always contains 0.
- GR 1 is a scratch register, often used as the implicit target of the ADDIL instruction.
- GR 2 is used for the return pointer (rp). A procedure must save its rp in the frame marker (see Chapter 5) before making any procedure calls. Leaf procedures do not need to save rp in the frame marker.
- GRs 3–18 are preserved across calls (all 64 bits of each).
- GRs 19–26 are used for parameter passing. See Chapter 5 for details. These are scratch registers.

- GR 27 serves as the global pointer (gp). This is a dedicated register with special conventions. Its use is subject to the following rules:
  a. On entry to a procedure, gp is guaranteed valid for that procedure.
  b. At any direct procedure call, gp must be valid (for the caller). This guarantees that an import stub (see Section 5.2) can access the linkage table.
  c. Any procedure call (indirect or direct) may destroy gp—unless the call is known to be local to the load module.
  d. At procedure return, gp must be valid (for the returning procedure). This allows us to optimize calls known to be local (the exceptions to Rule 'c').

  The effect of these rules is that gp must be treated as a scratch register at a point of call, but a procedure must preserve it from entry to exit.
- GRs 28 and 29 are used for return values up to 128 bits long. These are scratch registers.
- GR 29 is used as an argument pointer (ap) to point to the outgoing parameter area. This is a scratch register.
- GR 30 is used as the stack pointer (sp). It always points to the first byte beyond the end of the current (topmost) stack frame.
- GR 31 is used for (1) saving the return pointer in leaf procedures that make local millicode calls, or (2) the return pointer for external millicode calls.

4.2 Floating-Point Registers

The processor provides 32 double-precision floating-point registers, each 64 bits wide. For single-precision operation, the left and right halves of each may be individually addressed, providing 64 single-precision registers. A user-mode application may use all floating-point registers. The runtime architecture defines the following conventions for using these registers across normal procedure calls:

- FRs 0–3 contain the floating-point status word and exception registers.
- FRs 4–11 are used for parameter passing. See Chapter 5 for details.
- FRs 12–21 are preserved across calls.
- FRs 22–31 are scratch registers.

4.3 Other Registers

Space registers are controlled by the operating system. They may not be modified by user-mode applications, and should not be read.

The shift amount register (SAR) in CR 11 is a scratch register.

CR 27 is a read-only control register that is used as a thread-pointer. For kernel-thread implementations, the operating system controls the contents of this register, but a user-mode application may read it to access its thread-specific storage. For user-thread implementations, the operating system must provide an API-level routine for setting the value of this register.

All other registers defined by the processor architecture are reserved for use by the operating system and should not be read or written by a user-mode application.

5

Calling Conventions

This chapter describes the standard procedure calling conventions. These conventions include stack frame layout, the call and return mechanism, and parameter passing. Also covered in this chapter are the influences of dynamic linking on the calling conventions, and the special millicode calling convention.

The purpose of these calling conventions is to ensure that procedures compiled by different compilers, possibly written in different source languages, will interoperate correctly.

Compilers may define their own conventions for special-purpose calls, where the target of the call and its expectations are known to the compiler at the point of call. For example, a compiler may have its own compiler library, whose routines may be called only by compiler-generated code sequences; these calls are not necessarily subject to these standard conventions. In languages that allow nested procedure definitions, calls to nested procedures are considered special-purpose, and the compiler may define its own convention for providing access to the surrounding scopes.

The millicode library supplied by HP is an example of such a special-purpose calling convention; however, its interfaces are specified here so that third-party compilers may take advantage of this library.

5.1 Stack Frames

The operating system provides each process or thread its own stack. Stacks always grow toward higher addresses, and consist of a sequence of stack frames. Each frame corresponds to a procedure activation record in the call chain, the most recent activation record at the highest address. The stack pointer (sp) always points to the first byte beyond the top of the stack.

Stack frames must be 16-byte aligned, and must be a multiple of 16 bytes in length.

The stack frame layout is illustrated in Figure 5-1. Shaded regions identify areas of the stack frame that are written by called the called procedure rather than by the procedure creating the frame.

The stack frame contains the following regions:

- The register spill area. This area contains one 64-bit doubleword for each register spilled.
- The local storage area. This area contains storage for any temporaries or local variables needed by the procedure.
- The dynamic stack storage area. This is optional, and is created only by calls to alloca() or an equivalent routine.

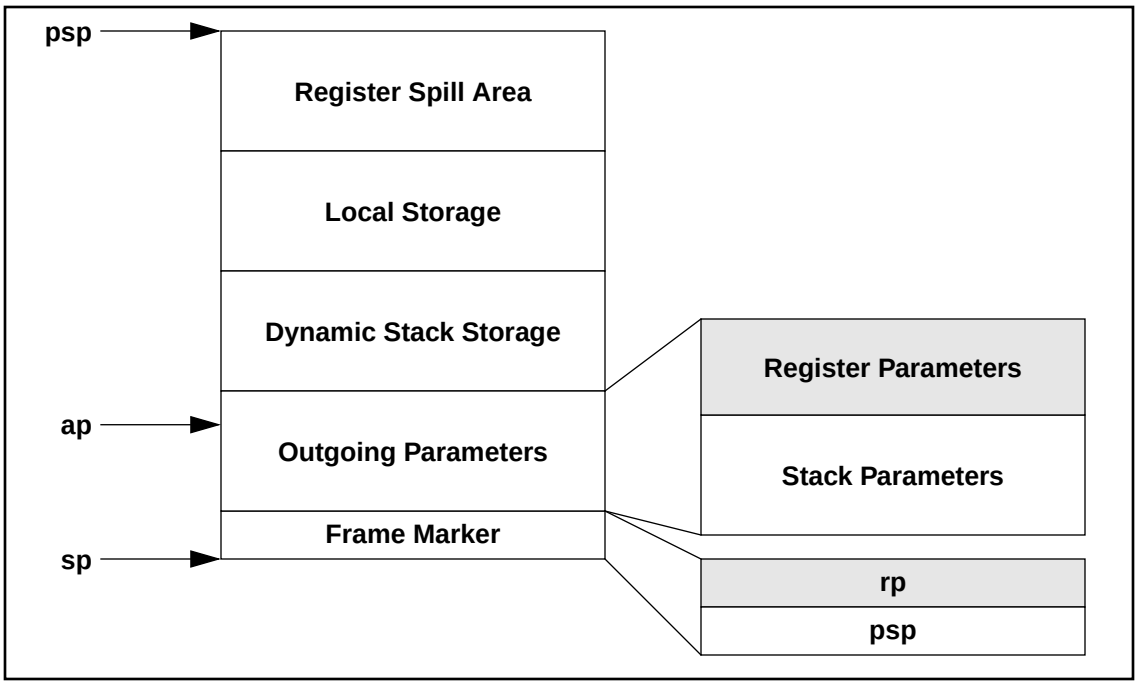

Figure 5-1. Stack Frame Layout

- The outgoing parameter area. This is a variable-size region below the frame marker (i.e., ending at sp–16). It is large enough to hold the outgoing actual parameters for any call made by the procedure. Parameters are allocated within this region from the lowest address to the highest address. Although the first 64 bytes of the parameter list are passed in registers, the parameter list must always allocate the storage for these parameters (the "home" locations), in case the called procedure needs to save the parameters to the stack (e.g., for varargs access). These 64 bytes must always be allocated, even if no call from the procedure passes that many actual parameters. The argument pointer (ap) points to the first byte of the stack parameters (i.e., the address of the beginning of the area plus 64). This area must be aligned at a 16-byte boundary.
- The frame marker. This consists of two 64-bit doublewords, which sometimes hold the saved return pointer (rp), at sp–16, and the previous stack pointer (psp), at sp–8.

The saved return pointer slot in the frame marker is used not by the procedure that creates the frame; it is reserved instead for use by any procedures that it calls. If a procedure needs to save its return pointer, it saves it in the return pointer slot in its caller's frame. Leaf procedures have no need to save their return pointer in the stack frame.

Because stack frames are typically fixed size, the previous stack pointer is normally not explicitly saved in the frame marker. Instead, it is calculated from the current stack pointer and the size of the frame. If the procedure has a variable-size frame (e.g., because it calls alloca() or equivalent), or if the frame is huge, the procedure may store the previous stack pointer in the frame marker.

5.2 Procedure Calls

This section describes the conventions for calling and returning. Sample code sequences may be found in Appendix A.

Direct calls

Direct procedure calls are made with the 22-bit displacement B,L instruction, with return pointer stored in GR 2. If the target is too distant (more than 8 MB in either direction) from the point of call, the linker will provide a long branch stub within reach of the point of call. The compiler must guarantee that the point of call can reach at least the beginning of the (input) section in which it is located. If the point of call is more than 8 MB from the beginning of the section, the compiler must generate long calls (see “Long Calls” in Appendix A).

Function pointers and indirect calls

A function pointer is a pointer to a function descriptor. The first two doublewords of the function descriptor are reserved, and should not be used. The third doubleword (at offset 16) in the descriptor is the entry point address, and the fourth doubleword (at offset 24) is the gp value. Every exported function has an “official” function descriptor defined in its load module, so that all function pointers that refer to the same function will have the same value. Non-exported functions whose addresses are materialized also have “official” function descriptors.

An indirect call loads the function pointer into a register. This value points to an official function descriptor, which contains the entry point address and a new gp value. The code loads the entry point address into a scratch register, and the new gp value into GR 27, then uses the BVE,L instruction to branch to the target, with the return pointer stored in GR 2.

DLL calls

Shared library (DLL) calls are made in two ways. Normally, the compiler has no information about whether a particular call will be resolved within the load module, so it will generate an ordinary direct call sequence as described above. The linker then recognizes that the target of the call is outside the load module, and resolves the call to an import stub, which is created at link time. The import stub accesses a local copy of the function descriptor, and loads the entry point address and new gp value, then branches to the target with a BVE instruction. An import stub does not store a return pointer because the original branch to the import stub will have already stored one.

If the compiler is given information that indicates a particular call is likely to be in a different load module, it may generate an external call sequence similar to the indirect call described above (see “Inline DLL Calls” in Appendix A).

Procedure returns

All procedure returns must use the BVE instruction, because the return point may be in a different space.

5.3 Parameter Passing

Parameters are first allocated in an conceptual argument list, which is then mapped to a combination of registers and stack memory. The argument list begins at relative address 0, and is allocated towards higher addresses.

Alignment and padding

Each parameter begins on a 64-bit (8-byte) boundary. Each 8-byte storage unit in the parameter list is called an argument slot. Some parameters use multiple argument slots.

Parameters are aligned and padded as follows:

- Integral scalar parameters smaller than 64 bits are padded on the left (with garbage) to 64 bits (i.e., the value is in the least-significant bits of the 64-bit storage unit, and the high-order bits are undefined).
- Single-precision (32-bit) floating-point scalar parameters are padded on the left with 32 bits of garbage (i.e., the floating-point value is in the least-significant 32 bits of a 64-bit storage unit).
- Double-extended-precision (80-bit) floating-point scalar parameters, if supported, are aligned on a 16-byte boundary, and padded on the right with 48 bits of garbage (i.e., the exponent and the most-significant 48 bits of the significand are placed in the first doubleword, and the least-significant 16 bits of the significand are placed in the high-order bits of the second doubleword).
- Quad-precision (128-bit) floating-point scalar parameters are aligned on a 16-byte boundary.
- Aggregates (structures and arrays, including complex numbers) smaller than 64 bits (8 bytes) are padded on the right (with garbage) to 64 bits (i.e., the value is in the most-significant bits of the 64-bit storage unit, and the low-order bits are undefined).
- Aggregates larger than 8 bytes are aligned on a 16-byte boundary, possibly leaving an unused argument slot, which is filled with garbage. If necessary, they are padded on the right (with garbage), to a multiple of 8 bytes.

Register parameters

The first 64 bytes of the parameter list are passed in registers, as follows:

- The first eight argument slots are associated, one-to-one, with general registers GR 26 (arg0) through GR 19 (arg7), and also with floating-point registers FR 4 (arg0) through FR 11 (arg7).
- Integral and aggregate parameters in this area are passed only in the corresponding general registers.
- If an aggregate parameter lies partly within this area, and partly beyond it, the part that lies within the first 64 bytes is passed in registers, and the remainder is passed in memory, as described below.
- Single- and double-precision floating-point parameters in this area are passed according to the available formal parameter information in a function prototype. If a prototype for the call is in scope, actual parameters matching floating-point formal parameters are passed in the corresponding floating-point registers. Actual parameters matching a variable-argument (...) specification in the prototype are passed in the corresponding general registers. If no prototype is in scope, floating-point parameters must be passed both in the corresponding general registers and in the corresponding floating-point registers.
- Single-precision floating-point parameters, when passed in floating-point registers, are passed in the right halves of the floating point registers; the left halves are unused.
- Double-extended- and quad-precision floating-point parameters within the first 64 bytes of the parameter list are always passed in general registers.

Stack parameters

The remainder of the parameter list, beginning with arg8 at relative address 64, is passed in the parameter area in the stack frame. Because the stack grows towards higher addresses, and the parameter list is allocated in the same direction, a pointer to arg8 (the beginning of the memory portion of the argument list) is placed in GR 29 (ap). This pointer must always be passed, even for calls with no memory parameters.

Variable argument lists

If there is no function prototype for the call, all floating-point arguments are passed in both GRs and FRs. If there is a function prototype, and some of the formal parameters are variable, the actuals corresponding to those formals are passed only in GRs. Otherwise, floating-point arguments are passed only in FRs, leaving the GRs mapped to those argument slots unused.

Passing aggregates by value

All aggregates passed by value are copied entirely into the argument list, regardless of size.

The convention for homogeneous floating-point aggregates (HFAs) is still under review. For at least the first two build environments, HFAs will be passed in the same manner as all other aggregates (i.e., in GRs only).

5.4 Return values

Integral return values are returned in GR 28. Values smaller than 64 bits are padded on the left (with garbage).

Single-precision and double-precision floating-point return values are returned in FR 4R (single precision) or FR 4 (double-precision). Double-extended- and quad-precision floating-point values are returned in GRs 28 and 29. The sign, exponent, and most-significant bits of the mantissa are returned in GR 28; the least-significant bits of the mantissa are passed in GR 29. For double-extended-precision values, GR 29 is padded on the right with 48 bits of garbage.

Aggregate return values up to 64 bits in size are returned in GR 28. Aggregates smaller than 64 bits are left aligned in the register; the pad bits on the right are undefined.

Aggregate return values between 65 and 128 bits are returned in GRs 28 and 29. The first 64 bits are placed in GR 28, and the remaining bits are placed, left aligned, in GR 29. The pad bits on the right of GR 29 (if any) are undefined.

Aggregate return values larger than 128 bits are returned in a buffer allocated by the caller. The address of the buffer must be passed in GR 28. The callee is not required to preserve the address in GR 28. The buffer must be aligned at a 16-byte boundary, and must not overlap any memory that may be otherwise visible to the called procedure.

5.5 Millicode calls

The runtime architecture defines two types of millicode calls: local millicode and external millicode. Local millicode calls assume that a copy of the called millicode routine is statically bound into the program; external millicode calls assume that the millicode routines are supplied as part of the operating system at a known address in global shared memory. External millicode is currently under consideration for use in a later release of the compilers.

Local millicode calls are made with the 22-bit displacement B,L instruction, with return pointer stored in GR 2. Leaf procedures must copy their return pointer from GR 2 to GR 31 if they make local millicode calls. As for normal procedure calls, the linker will generate a long branch stub if the target is too far from the point of call. If the point of call is more than 8 MB from the beginning of the input section, the compiler must generate a long call.

External millicode calls will use the BE,L instruction, using GR 0 and an as-yet-undetermined SR, storing the return pointer in GR 31. The absolute target address will be determined at link time.

5.6 Dynamic Linking

The operating system loader (exec) will load an executable file's loadable segments prior to invoking a program interpreter (dld).

The runtime architecture does not require the use of export stubs. Import stubs and indirect call sequences are responsible for loading the gp value from the function descriptor, and the BVE instruction used for procedure returns will always return to the proper space.

6

Stack Unwinding and Exception Handling

Stack unwinding is the process of tracing backwards through a process' stack of activation records. In order to print a stack trace, debuggers require the ability to identify every frame on the stack, and to show the process context associated with each one. Exception handling often requires the ability to remove a number of frames from the stack and to transfer control to an exception handling routine that may have been far down the stack.

While different procedures will need differently-sized stack frames, we expect that most procedures will allocate a frame whose size does not change while the procedure is active. Thus, for most procedures, we can simply record this fixed frame size in a static table, and use the instruction address (IA) as a key to this table. For procedures whose frames can vary in size, we must impose a convention for saving and recovering the sp value for the previous frame on the stack.

As the stack is unwound, it is also necessary to recover the values of preserved registers that were saved by each procedure in the activation stack, so that debuggers have access to correct values of local variables, and so that exception handlers can operate correctly. This requirement also imposes conventions for saving and recovering the values of these preserved registers.

In all cases, we wish to retain as much flexibility as possible for the compiler in its use of registers and code generation. Thus, these conventions allow the compiler to save the necessary values in a variety of locations, and with a variety of code sequences. We use the IA as a key for locating an unwind table entry that describes everything necessary for locating the previous stack frame, as well as the previous IA. The compiler is responsible for generating this static unwind table entry for each procedure that it generates code for.

On most platforms, unwinding the stack will be done via an unwind library that can be called from the process itself, from a debugger, or for exception handling. It operates on context records; the primary routine reconstructs the context for a previous frame given the context for its descendent frame. Because the structure of a context record, and the interface between the OS and exception handling mechanism is platform dependent, this unwind library is also platform-dependent, and is not defined as part of the runtime architecture. This chapter describes the framework for unwinding the stack and for processing exceptions, including the format of the static unwind tables constructed by the compilers, and the code generation conventions imposed as a result.

6.1 Unwinding the stack

The process of unwinding the stack begins with an initial context record describing the process state in the most recent procedure activation, at the point of interruption. From this initial state, the stack is unwound one procedure frame at a time, using static information generated by the compilers about

each procedure to help it reconstruct a context record describing the previous procedure, which is suspended at a point just after the procedure call or an asynchronous interruption.

Initial context

Every stack unwind starts with an initial context, obtained from one of three sources:

- The debugger. The context record is obtained from the OS through the debugging API.
- The unwind library. The context is constructed from the state of the current process.
- From exception handler. The context is constructed by the OS and passed to the exception handler.

Step to previous frame

This process builds a context record corresponding to the next older frame on the stack. This context record can, in turn, be used to unwind to the next frame. The following steps will reconstruct the context for the previous frame:

1. Find the return link in the current context, and set IA in the previous context to that address.
2. Set sp in previous context to sp from current context minus the current memory frame size.
3. Find the saved copies of the preserved registers in the current context, and copy them to the previous context.

The information needed to execute these steps correctly is recorded by the compilers in static unwind information, stored in the text segment of the program itself. The structure of this information is described in Section 6.3. Each text segment contains a table of unwind information, and the dynamic loader is expected to provide an API for finding the unwind table, given a known IA. This API is specific to the operating environment, and is not described here.

When a process is delivered an asynchronous interruption (via a mechanism that is platform dependent), the full process context needs to be saved so that the process can continue executing correctly once the interruption has been handled. Typically, this context will be saved on the memory stack, and a new procedure frame will be constructed for the interruption handler. The first procedure frame in the interruption processing must be marked in such a way that the unwind routine can recognize that unwinding past the point of interruption requires a restoration of the full context. This, unfortunately, is also a platform-dependent operation, and cannot be described in the runtime architecture.

6.2 Exception handling framework

The exception handling model for PA-RISC 2.0 is partitioned into a language-independent component and a language-dependent component. The language-independent component is responsible for fielding an exception, searching for an exception handler, and unwinding the stack prior to processing an exception. Each source language that supports exception handling must provide, as part of its runtime library, a “personality” routine that implements the language-dependent component of this model.

This document uses the C++ exception handling mechanism as an example of the language-dependent component. The description of the C++-specific data structures and routines should be treated as an example, rather than a specification of the C++ design. Text that discusses language-specific implementation appears indented and italicized like this paragraph.

The exception handling model is oriented around procedure frames on the memory and register stacks. Each frame corresponds to an activation of a procedure, which may or may not have associated

exception handling requirements. A procedure may have two kinds of exception handling requirements:

- It may allocate some objects that require deallocation or some other form of cleanup if the procedure or any of its blocks are terminated abnormally.
- It may have one or more *try regions*, which are regions of code that specify an action to be taken if an exception occurs while control is within them.

In either of these cases, the compiler records the requirements in the static unwind information for the procedure, and stores a reference to the personality routine for that procedure. Typically, a language will use a single personality routine for all procedures, but this is not a requirement (for example, a language may define a separate personality routine for procedures that require cleanup, but have no try regions.)

Try regions may be nested both statically, within the procedure, and dynamically, through procedure calls. When an exception occurs, each try region is inspected to determine if it has specified an action for that particular exception. The try regions are inspected in order, beginning with the innermost region.

> *In C++, a* try/catch *statement defines a try region, and the catch clause controls which exceptions are to be caught and handled within that region.*

Exceptions are raised by invoking a routine in the language-independent component called the *exception dispatcher*, which initiates the process of handling the exception. Synchronous exceptions may be raised directly by the application through a language-specific construct; asynchronous exceptions may be raised in response to hardware-detected traps or faults.

> *In C++, synchronous exceptions can be raised with the* throw *statement. This statement creates an exception object, which is matched against the prototype in each* catch *clause for each active* try *statement. C++ does not define asynchronous exceptions.*

The dispatcher unwinds each frame on the stack non-destructively, beginning with the topmost frame, searching for frames with one or more try regions. For each frame that has exception handling information, the dispatcher invokes the personality routine, which determines which try regions, if any, are currently active. For each active try region, starting with the most deeply nested one, the personality routine determines whether to dismiss the exception, handle it, or continue the search with the next try region, or with the previous frame on the stack. If the personality routine does find a try region with a handler for the exception, it invokes the unwinder to unwind the stack a second time. During this second unwind, the unwinder invokes the personality routines for each frame again so that cleanup actions may be executed as necessary. When the unwind reaches the frame that contains the exception handler, control is tranferred to the handler.

Because the exception handler may need to access exception information that is on the memory stack, the handler executes before any frames are removed from the stack. This means that the handler must execute as a “nested” procedure, and access to local variables in the enclosing procedure must be made through a static link. Although cleanup actions have been performed for all frames that have been unwound, those stack frames are not physically popped from the stack until the exception handler finishes.

Because the global pointer (gp) is not guaranteed to be valid at an arbitrary point in a procedure, the personality routines and the unwinder must set a valid gp prior to transferring control to any exception filters, cleanup routines, or exception handlers.

The relationships among these components are illustrated in Figure 6-1. The shaded boxes identify the components that are specific to C++.

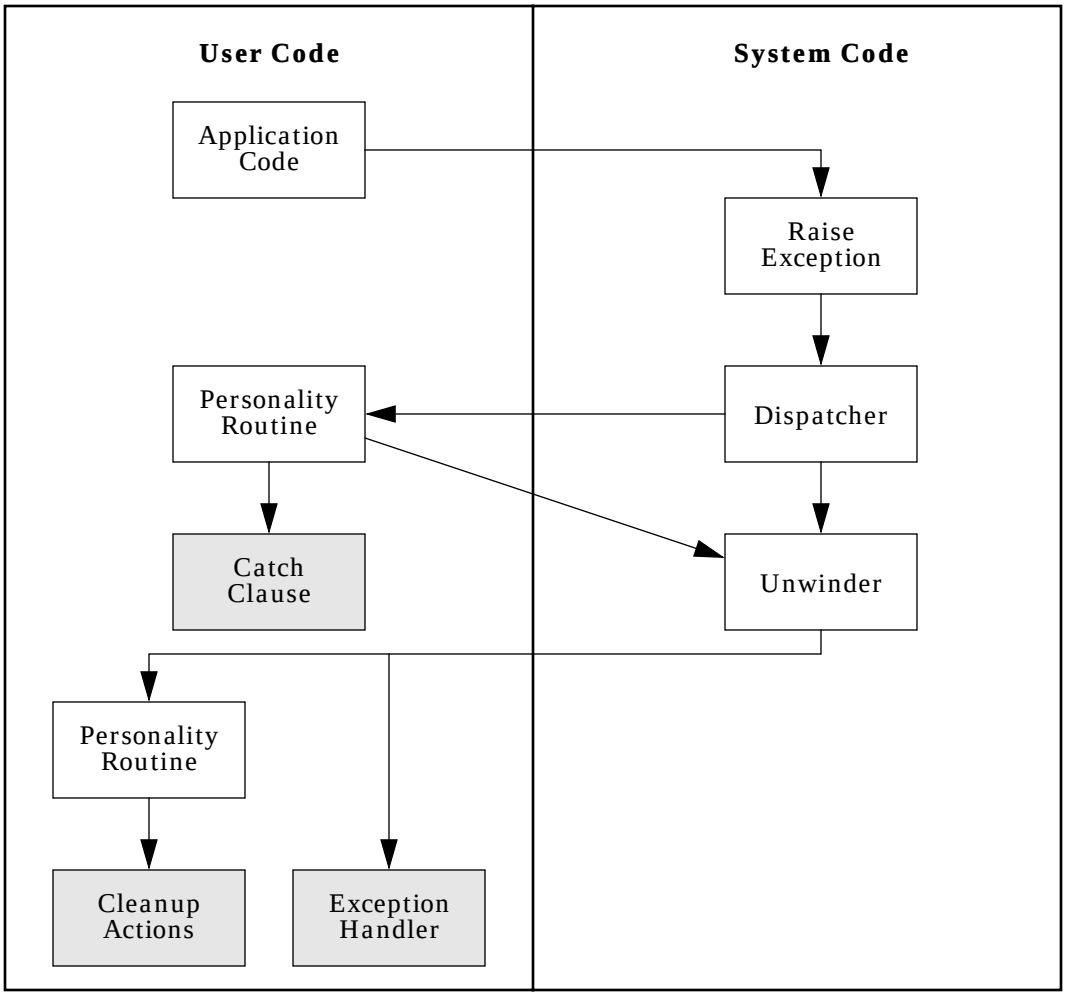

Figure 6-1. Components of the exception handling mechanism

6.3 Unwind Data Structures

The unwind data structures will remain the same as in the 32-bit runtime, with one exception: The procedure start and end addresses in each unwind descriptor will be segment-relative offsets instead of absolute link-time addresses. These offsets may be generated using segment-relative relocations; the segment base should be specified using the section symbol for the .btext section. This section will be defined by the linker as the first section in the text segment.

The following bit fields in the unwind flags will not be used in the 64-bit runtime, and will become reserved:

- millicode_save_sr0
- sr4export
- variable_frame
- separate_package_body

- frame_extension_millicode
- save_r19
- mpe_xl_interrupt_marker

NOTE: Version 3.0 of this document listed the entry_sr bit as reserved. This bit will still be used to indicate that SR 3 has been preserved. Although user-mode applications cannot use the space registers, kernel code can and will still need this bit.

A new unwind flag will be used to indicate leaf procedures that have moved the return pointer to r31 in order to make local millicode calls. The instruction to copy r2 to r31 must appear in the procedure prologue at the same point where a store instruction to save rp in the stack would appear. The return pointer must remain valid in r31 through the end of the procedure.

The millicode_save_sr0 bit will be renamed rp_in_r31, and used to indicate such leaf procedures.

Frames up to 32K (2^{15}) bytes in size may be allocated in a single (LDO) instruction.

There will be no stub unwind table because there will not be any stubs that require unwind information.

The unwind library must obtain load module information from the dynamic loader for each IA value that it gets. The load module information contains the addresses of the unwind tables and the gp value for that load module.

6.4 Coding Conventions for Proper Unwinding

In order to guarantee the unwindability of a procedure from any point in its code, the compilers and assembly language programmers must observe certain conventions with regard to the entry and exit sequences in a procedure.

The .enter and .leave directives in the assembler produce entry and exit code that obey these conventions.

These conventions will be described in a future release of this document.

7

System Interfaces

This chapter covers a few aspects of the interfaces between an application and the operating system that are general in nature, and not specific to any single operating system.

7.1 Process Initialization

An application begins its execution at a specified program entry point, which depends on the primary language in which the application is written. For C programs, the function main is the program entry point. On most operating systems, however, some system-dependent initialization must take place before control is transferred to this entry point. This initialization may take place in the OS, in start-up code that must be linked with the application, or in the DLL loader.

This section presents a general overview of what an application expects when its program entry point receives control. The ABI document for each operating system is expected to contain the details.

Initial Memory Stack

The memory stack pointer, sp, must be properly aligned, and must contain an address that is suitable for allocation of the program's first stack frame.

Initial Register Values

The sp and gp registers must be initialized correctly, sp as described above and gp to the global pointer value for the main program's short data segment. The tp register must also be initialized to point the the thread data segment for the main thread of execution.

The contents of the argument registers are system-dependent, and are typically used for transmitting the program arguments and other information about the environment.

7.2 System Calls

System API routines are called using the standard calling conventions described in Chapter 5. Any special interfaces between these API routines and the operating system itself is system-dependent, and these API routines are typically supplied in a system DLL.

7.3 Traps and signals

When the OS delivers a signal or an exception to a user process, it must make the following available to the process:

- A context record, containing the full user-visible context.
- The cause of the trap. If the trap was caused by an instruction, the information must be sufficient to identify the bundle and syllable.

When a trap or signal handler returns, OS help is necessary for restoring the complete context (via RFI). Thus, the OS must build a dummy stack frame for the handler, so that a return from the handler will transfer to an OS entry point that can restore the full context (this is sigreturn on HP-UX).

Trap handlers may also need to modify the state of the registers before returning to the interrupted code, and may need to nullify the execution of the instruction that caused the trap.

Appendix A

Code Examples

A.1 Literals

Literals and other read-only local data in the text segment may be loaded using pc-relative addressing or through a linkage table entry. The pc-relative code sequence is as follows:

```
$L1:    MFIA    tmp
        ...
        ADDIL   L'literal+(.–$L1)–., tmp        ; PCREL21L
        LDD     R'literal+(.–$L1)–.(r1), tgt    ; PCREL14DR
```

(These two instructions get a pc-relative relocation type, where the "(.–$L1)" part of the expression is the constant addend.)

The sequence using the linkage table is as follows:

```
        LDD     T'literal(gp), tmp      ; LTOFF16F
        LDD     0(tmp), tgt
```

The second sequence has the advantage that the first LDD can be shared by many accesses to the procedure's literal pool. The compiler can put all literals for the procedure in one block with a single symbol, load that symbol's address from the linkage table once, then load each literal with an offset relative to that address.

In the first sequence, the ADDIL cannot be shared for separate literals because of problems with the L/R rounding mode; the LR/RR rounding mode cannot be used because the addend on the LDD may round differently from the addend on the ADDIL.

A.2 Accessing Data

Data with external scope and all data in the huge region must be addressed through the linkage table:

```
        ADDIL   LT'var, gp              ; LTOFF21L
        LDD     RT'var(r1), tmp         ; LTOFF14DR
        LDD     0(tmp), tgt
```

Other data that is local to the load module and not subject to pre-emption ("own" data) may be addressed relative to gp:

```
        ADDIL   LR'var–$global$, gp             ; GPREL21L
        LDD     RR'var–$global$(r1), tgt        ; GPREL14DR
```

A.3 Accessing Thread-Local Storage

Items in thread-local storage must be accessed by first loading a tp-relative offset from the linkage table, then adding this offset to the tp register:

```
MFCTL     cr27, tmp1
...
ADDIL     LT'var–__tp, gp              ; LTOFF_TP21l
LDD       RT'var–__tp(r1), tmp2        ; LTOFF_TP14DR
ADD       tmp1, tmp2, tmp3
LDD       0(tmp3), tgt
```

Because of the high latency for a move from control register instruction, the MFCTL should be executed only once in any procedure that accesses thread-local storage, and the target GR should be reused.

A.4 Direct Procedure Calls

Direct short calls use the long-displacement form of the B,L instruction:

```
COPY      gp, save_gp
...
B,L       target, r2                   ; PCREL22F (BLL)
<delay slot>
COPY      save_gp, gp
```

A.5 Indirect Procedure Calls

Indirect calls should be compiled entirely inline:

```
COPY      gp, save_gp
...
(assume function pointer is in register “fp”)
LDD       16(fp), tmp
LDD       24(fp), gp
BVE,L     (tmp), r2
<delay slot>
COPY      save_gp, gp
```

A.6 Inline DLL Calls

If the compiler is told a direct call is to another load module, the call should be compiled like an indirect call, but it may avoid an extra indirection by using a local copy of the function descriptor. (This sequence is always safe, even if the target is in the same load module.)

```
COPY      gp, save_gp
...
ADDIL     LRQ'function, gp             ; PLTOFF21L
LDD       RRQ'function(r1), tmp        ; PLTOFF14DR
LDD       RRQ'function+8(r1), gp       ; PLTOFF14DR
BVE,L     (tmp), r2
<delay slot>
COPY      save_gp, gp
```

A.7 Millicode Calls

Local millicode calls use the long-displacement form of the B,L instruction:

```
        B,L     mroutine, r2            ; PCREL22F
        <delay slot>
```

External millicode calls use the BE,L instruction:

```
        BE,L    mroutine(r0,sr?), r31   ; DIR17F
        <delay slot>
```

There is currently no plan to support external millicode; the space register to be used for this call is TBD.

A.8 Function Pointers

Function pointers should be materialized by loading them from the DLT:

```
        ADDIL   LTP'function,gp         ; LTOFF_FPTR21L
        LDD     RTP'function(r1), tgt   ; LTOFF_FPTR14DR
```

To initialize a variable to the address of a function in the PIC model, the compiler or assembler should use the P' operator to obtain a plabel:

```
var:    .dword  P'function              ; FPTR64
```

A.9 Long Calls

The compiler should *not* generate long calls by default.

Long calls can be done using a long pc-relative sequence, or by making the call indirect through the linkage table. The long pc-relative sequence is as follows:

```
        COPY    gp, save_gp
$L1:    MFIA    tmp
        ...
        ADDIL   L'function+(.–$L1)–., tmp       ; PCREL21L
        LDO     R'function+(.–$L1)–.(r1), tmp2  ; PCREL14R
        BVE,L   (tmp2), r2
        <delay slot>
        COPY    save_gp, gp
```

The MFIA instruction can be shared with literal accesses as well.

The indirect alternative is the same as the inlined DLL call.

```
        ADDIL   LT'var–__tp, gp         ; LTOFF_TP21l
        LDD     RT'var–__tp(r1), tmp2   ; LTOFF_TP14DR
```

A.10 Optimizations

The compiler may choose to implement a “small” PIC option, in which the linkage table is restricted to 8192 entries. Under this option, all references to the linkage table may be made with a single LDD instruction using a T' reference, replacing the ADDIL/LDD pair. It may also implement a similar option that applies to all own data, where it assumes that the entire short data region is no larger than 64K, as described in Section 2.4.

If a global data item is known to be defined in the same load module and not subject to pre-emption, it may be accessed as “own” data. The +Onoextern compiler option may be used to designate such symbols, and is generally safe for code that will be placed in the main program.

When a program is known to be statically-bound (i.e., no DLLs are bound in), the tp-relative offset can be calculated statically at link time, so a more direct sequence for thread-local storage may be used:

```
MFCTL     cr27, tmp1
…
ADDIL     LR'var–__tp, tmp1          ; TPREL21L
LDD       RR'var–__tp(r1), tgt       ; TPREL14DR
```

When the target of a call is known to be in the same load module as the callee, the call may assume that gp will not be destroyed, and does not need to restore gp after the call. (The +Onoextern compiler option may be used to designate such target symbols.)

When indirect calls are known to stay within a single load module (and the function pointers are not themselves passed to another load module), the setting and restoring of the gp register may be omitted:

```
(assume function pointer is in register “fp”)
LDD       16(fp), tmp
BVE,L     (tmp), r2
<delay slot>
```

This optimization is enabled by the +ESfic compiler option.

When code is being compiled for a statically-bound program, the effects of the +Onoextern and +ESfic options may be applied globally.

Appendix B

Standard Header Files

B.1 Implementation Limits

The following constants are defined in the <limits.h> header file.

```
#define CHAR_BIT                8
#define SCHAR_MIN               (-128)
#define SCHAR_MAX               127
#define UCHAR_MAX               255

/* MB_LEN_MAX determined by locale information */

#define CHAR_MIN                SCHAR_MIN
#define CHAR_MAX                SCHAR_MAX

#define SHRT_MIN                (-32768)
#define SHRT_MAX                32767
#define USHRT_MAX               65535

#define INT_MIN                 (-2147483647-1)
#define INT_MAX                 2147483647
#define UINT_MAX                4294967295

#define LONG_MIN                (-9223372036854775807-1)
#define LONG_MAX                9223372036854775807
#define ULONG_MAX               18446744073709551615
```

B.2 Floating-Point Definitions

The following constants are defined in the <float.h> header file.

```
#define FLT_DIG                 6                   /* Max (decimal) digits of precision */
#define FLT_EPSILON             1.19209290e-07F
#define FLT_MANT_DIG            24
#define FLT_MAX                 3.40282347e+38F
#define FLT_MAX_10_EXP          38
#define FLT_MAX_EXP             128
#define FLT_MIN                 1.17549435e-38F
#define FLT_MIN_10_EXP          (-37)
#define FLT_MIN_EXP             (-125)
#define FLT_RADIX               2
```

```
#define FLT_ROUNDS              1
#define FLT_GUARD               0
#define FLT_NORMALIZE           0

#define DBL_DIG                 15          /* Max (decimal) digits of precision */
#define DBL_EPSILON             2.2204460492503131e-16
#define DBL_MANT_DIG            53
#define DBL_MAX                 1.7976931348623157e+308
#define DBL_MAX_10_EXP          308
#define DBL_MAX_EXP             1024
#define DBL_MIN                 2.2250738585072014e-308
#define DBL_MIN_10_EXP          (-307)
#define DBL_MIN_EXP             (-1021)

#define __FLOAT80_DIG           18          /* Max (decimal) digits of precision */
#define __FLOAT80_EPSILON       1.08420217248550443400 75E–19L
#define __FLOAT80_MANT_DIG      64
#define __FLOAT80_MAX           1.18973149535723176505e+4932L
#define __FLOAT80_MAX_10_EXP    (+4932)
#define __FLOAT80_MAX_EXP       (+16384)
#define __FLOAT80_MIN           3.36210314311209350626e–4932L
#define __FLOAT80_MIN_10_EXP    (–4931)
#define __FLOAT80_MIN_EXP       (–16381)
```

B.3 Variable Argument List Macros

The following definitions roughly define the operation of the variable argument list macros provided in the <stdarg.h> header file. Similar definitions for K&R C may be found in <varargs.h>.

```
typedef char *va_list;

#define _VA_ALIGN(list, align)                                          \
    (va_list)(((unsigned int)(list) + (align) – 1) & ~((align) – 1))

#define va_start(list, parmN)       (list = (va_list)(&parmN + 1))

#define va_arg(list, mode) (                                            \
    list = _VA_ALIGN(list, ((sizeof(mode) > 8) ? 16 : 8)) +             \
        ( ((sizeof(mode) < 8) && !__is_aggregate(mode)) ?               \
            8 – sizeof(mode) : 0 ),                                     \
    *(mode *)list++                                                     \
    )
```

The va_arg macro requires the built-in __is_aggregate() function in the compiler; it returns true if the type given as the argument is an aggregate type.

B.4 Setjmp/Longjmp

The following definitions are provided in the <setjmp.h> header file.

```
#define _JBLEN                  40
typedef long double jmp_buf[_JBLEN];
```

The jump buffer is defined to be long enough to contain the context that must be saved at a procedure call boundary, and includes additional space reserved for future use. The contents include the following:

- Stack pointer (sp)
- All preserved general and floating-point registers (GRs 3–18 and FRs 12–21)
- The instruction address (the return pointer from the call to setjmp)
- Saved signal mask

The locations of individual items within the jump buffer is ABI specific.