Power Architecture® 32-bit Application Binary Interface Supplement 1.0 - Embedded

Ryan S. Arnold
IBM

Greg Davis
Green Hills

Brian Deitrich
Freescale Semiconductor

Michael Eager
Eager Consulting

Emil Medve
Freescale Semiconductor

Steven J. Munroe
IBM

Joseph S. Myers
CodeSourcery

Steve Papacharalambous
Freescale Semiconductor

Anmol P. Paralkar
Freescale Semiconductor

Katherine Stewart
Freescale Semiconductor

Edmar Wienskoski
Freescale Semiconductor

Power Architecture® 32-bit Application Binary Interface Supplement 1.0 - Embedded

by Ryan S. Arnold, Greg Davis, Brian Deitrich, Michael Eager, Emil Medve, Steven J. Munroe, Joseph S. Myers, Steve Papacharalambous, Anmol P. Paralkar, Katherine Stewart, and Edmar Wienskoski

1.0 Edition
Published April 19, 2011

The ATR-LINUX portions of this document are derived from the 64-bit PowerPC ELF Application Binary Interface Supplement 1.8, originally written by Ian Lance Taylor under contract for IBM, with later revisions by: David Edelsohn, Torbjorn Granlund, Mark Mendell, Kristin Thomas, Alan Modra, Steve Munroe, and Chris Lorenze.

The ATR-TLS and ATR-SECURE-PLT sections of this document are original contributions of IBM written by Alan Modra and Steven Munroe.

The ATR-SPE and ATR-EABI portions of this document are derived from material used to write the E500 ABI and are contributed by Freescale Semiconductor.

Revision History
Revision 1.0 April 19, 2011 Revised by: Power.org PowerABI TSC

Table of Contents

List of Figures

List of Tables

Preface

1. How To Read This Document

Implementations of this *Power Architecture 32-bit Application Binary Interface Supplement* should indicate which *ABI software features* (see *Appendix A)* and Power ISA™ *categories* are implemented. When reading this document, the reader should reference those constraints and selectively read this text based upon them.

Appendix A provides a taxonomy of the information in this ABI document. The core of the ABI is common to all implementations and appears as nonconditional text, tables, and graphics.

Optional *ABI software feature* text or Power ISA *category* specific text is represented in the taxonomy as conditional attributes of the form **ATR-*FOO*** (where "*FOO*" is one of the attributes described in *Appendix A*). These attributes are used in the ABI text as element tags which aid in selective reading (and the generation) of this ABI document. These attributes describe the relationship of the optional elements of this document to a specific implementation.

This version of the *Power Architecture 32-bit Application Binary Interface Supplement* may take one of the following forms:

Linux & Embedded

The unified ABI document contains all text from all implementations of the ABI.

Linux

The technical conditions governing implementations of the Linux ABI are described by attribute conformance and inclusion rules in *Appendix B*, *Section B.1*. The attribute tags described in that part of the appendix are used to conditionally generate the Linux ABI variant of this document.

Embedded

The technical conditions governing implementations of the Embedded ABI are described by attribute conformance and inclusion rules in *Appendix B*, *Section B.2*. The attribute tags described in that part of the appendix are used to conditionally generate the Embedded ABI variant of this document.

Document elements representing *Categories* of the Power ISA are required for a software implementation based upon the implementation's conformance with either *Book III-S* or *Book III-E* of the Power ISA.

The following bounding box exemplifies a document element which corresponds to a *category* of the Power ISA.

ATR-SPE

This is an example of conditional text that applies to implementations that support the Signal Processing Engine (SPE) ABI, an optional *category* of the Power ISA.

This document also contains elements that correspond to optional *ABI software features* that may or may not be present in specific implementations. A prime differentiation would be software features used in embedded environments vs. those used in server environments, e.g., support for threading as defined by the Thread Local Storage ABI, support for the secure-PLT, or support for dynamic linking.

!ATR-TLS

This is an example of conditional text that applies to an implementation which **does not** support a specific software feature.

2. Section Numbering

The subsection numbering of the unified *Linux & Embedded* version of the *Power Architecture 32-bit Application Binary Interface Supplement* is sequential and does not skip digits between sibling subsections since it contains all of the text, tables, and graphics available.

The individual *Linux* and *Embedded* versions of the *Power Architecture 32-bit Application Binary Interface Supplement* contain a subset of the text, tables, and graphics available. The subsection numbers of these subset documents remain congruent with those of the *Linux & Embedded* version of the *Power Architecture 32-bit Application Binary Interface Supplement* (and with each other where they overlap) in order to prevent confusion during cross-reference and therefore subsection numbering can appear to skip digits between sibling subsections.

Chapter 1. Introduction

The *Executable and Linkable Format* (ELF) defines a linking interface for executables and shared objects in two parts. The first part is the generic System V ABI. The second part is a processor-specific supplement.

This document is the processor-specific supplement for use with ELF on 32-bit Power Architecture processor systems. This is not a complete System V Application Binary Interface Supplement because it does not define any library interfaces.

Furthermore, this document establishes both big-endian and little-endian application binary interfaces (see Section 3.1.2.1). Processors in the 32-bit Power Architecture can execute in either big-endian or little-endian mode. Executables and executable generated data (in general) that subscribe to either byte ordering are not portable to a system running in the other mode.

Note: This ABI specification does not address little-endian byte ordering prior to Power ISA 2.03.

The *Power Architecture 32-bit Application Binary Interface Supplement* is not the same as the *64-bit PowerPC ELF ABI*.

The *Power Architecture 32-bit Application Binary Interface Supplement* is intended to use the same structural layout now followed in practice by other processor specific ABIs.

1.1. Reference Documentation

The archetypal ELF ABI is described by the *System V ABI*. Supersessions and addenda that are 32-bit Power Architecture processor-specific are described in this document.

The following cited documents are complementary to this document and equally binding:

- *Power Instruction Set Architecture Version 2.05*, IBM, 2007. http://www.power.org/resources/reading/PowerISA_V2.05.pdf
- *DWARF Debugging Information Format Version 4*, DWARF Debugging Information Format Workgroup, 2010. http://dwarfstd.org/Dwarf4Std.php
- *ISO/IEC 9899:1999(E): Programming languages—C*, as amended by ISO/IEC 9899:1999/Cor.1:2001(E), ISO/IEC 9899:1999/Cor.2:2004(E) and ISO/IEC 9899:1999/Cor.3:2007(E), http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf

ATR-SPE

- *SPEPIM: Signal Processing Engine Auxiliary Processing Unit Programming Interface Manual*, Freescale Semiconductor, 2004. http://www.freescale.com/files/32bit/doc/ref_manual/SPEPIM.pdf?fsrch=1

ATR-VLE

- *VLEPEM: Variable-Length Encoding (VLE) Programming Environments Manual*, Freescale Semiconductor, 2007. http://www.freescale.com/files/32bit/doc/ref_manual/VLEPEM.pdf?fsrch=1

The following documents are of interest for their historical information but are not normative in any way.

- *The [32-bit] PowerPC Processor Supplement*, Sun Microsystems, 1995.
- *The 32-bit AIX ABI.*
- *The PowerOpen ABI.*

Chapter 2. Software Installation

2.1. Physical Distribution Media and Formats

This document does not specify any physical distribution media or formats. Any agreed-upon distribution media may be used.

Chapter 3. Low Level System Information

3.1. Machine Interface

3.1.1. Processor Architecture

This Application Binary Interface (ABI) is not explicitly predicated on a minimum Power ISA version.

All nonoptional instructions that are defined by the Power Architecture® can be assumed to be implemented and work as specified. ABI conforming implementations must provide these instructions through software emulation if they are not provided by the processor.

> Note: The exceptions to this rule are the *Fixed-point Load and Store Multiple* and *Fixed-point Move Assist* instructions which are not available in little-endian implementations because they would cause alignment exceptions.

Processors may support additional instructions beyond the published Instruction Set Architecture (ISA) and the Power Architecture optional ones, through *Auxiliary Processing Units* (APUs). This ABI provides a method for describing the additional instructions in section information (see *Section 4.4* and *Section 4.10*) but does not address these additional instructions directly and executing them may result in undefined behavior.

This ABI does not explicitly impose any performance constraints on systems.

3.1.2. Data Representation

3.1.2.1. Byte Ordering

The following standard data formats are recognized:

- 8-bit byte
- 16-bit halfword
- 32-bit word
- 64-bit doubleword
- 128-bit quadword

In big-endian byte ordering, the most significant byte is located in the lowest addressed byte position in memory (byte 0). This byte ordering is alternately referred to as *Most Significant Byte* (MSB) ordering.

In little-endian byte ordering, the least significant byte is located in the lowest addressed byte position in memory (byte 0). This byte ordering is alternately referred to as *Least Significant Byte* (LSB) ordering.

A specific processor implementation must state which type of byte ordering is to be used.

ATR-SPE

Although it is possible on some processors to map some pages as little-endian, and other pages as big-endian in the same application, such an application does not conform to the ABI.

Table 3-1, *Table 3-2, Table 3-3*, and *Table 3-4* show the conventions being assumed in big-endian and little-endian byte ordering at the bit and byte levels. These conventions are applied to integer and floating-point data types. Byte numbers are indicated in the upper corners, and bit numbers in the lower corners. Little-endian byte numbers are indicated on the right side; big-endian byte numbers are indicated on the left side.

Table 3-1. Bit and Byte Numbering in Halfwords

0	1	1	0
msb		lsb	
0	7	8	15

Table 3-2. Bit and Byte Numbering in Words

0	3	1	2	2	1	3	0
msb						lsb	
0	7	8	15	16	23	24	31

Table 3-3. Bit and Byte Numbering in Doublewords

0	7	1	6	2	5	3	4
msb							
0	7	8	15	16	23	24	31
4	3	5	2	6	1	7	0
						lsb	
32	39	40	47	48	55	56	63

Table 3-4. Bit and Byte Numbering in Quadwords

0	15	1	14	2	13	3	12
msb							
0	7	8	15	16	23	24	31
4	11	5	10	6	9	7	8
32	39	40	47	48	55	56	63
8	7	9	6	10	5	11	4
64	71	72	79	80	87	88	95
12	3	13	2	14	1	15	0
						lsb	
96	103	104	111	112	119	120	127

Note: In the Power ISA, the figures are generally only shown in big-endian byte order. The bits in these data format specification are numbered from left to right (MSB to LSB).

ATR-SPE

Note: SPE documentation uses 64-bit numbering throughout, including for registers such as the CR that only contain 32 bits. This numbering can lead to some confusion. For example, although the CR bits are now numbered from 32 to 63, the same assembly instructions still work: `crxor 6,6,6` operates on bit $32 + 6$, that is, CR[38]. When discussing register contents, the bits are numbered 0 : 63 for 64-bit registers and 32 : 63 for 32-bit registers. When discussing memory contents, the bits are numbered naturally (for example, 0 : 7 for bits within one byte and 0 : 15 for bits within halfwords).

The bit numbering in the Power ISA is all 64-bit except for the following registers indicated in Power ISA section 1.4:

- Opcodes marking 0-31

ATR-CLASSIC-FLOAT

- As of Power ISA version 2.05 the FPSCR has been extended from 32-bits to 64-bits. The fields of the original 32-bit FPSCR are now held in bits 32-63 of the 64-bit FPSCR. The assembly instructions which operate upon the 64-bit FPSCR have either had a *W Instruction Field* added to select the operative word for the instruction, e.g., `mtfsfi`, or the instruction has been extended to operate upon the entire 64-bit FPSCR, e.g., `mffs`. Reference to fields of the FPSCR, representing 1 or more bits, is done by field number with an indication of the operative word rather than by bit-number.

  If the Power ISA version 2.05 DFP category is not needed by an implementation the FPSCR may continue to be referenced as a 32-bit register using the old forms of the instructions to support binary compatibility of ELF files built against an older Power ISA version. See *Section 3.2.1* for more information on the FPSCR.

3.1.2.2. Fundamental Types

The following tables map the data format specifications described in the Power ISA to ISO C scalar types. Each scalar type has a required alignment, which is indicated in the alignment column. Usage of these types in data structures must follow the alignment specified in the order encountered to ensure consistent mapping. When using variables individually, more strict alignment may be imposed if it has optimization benefits.

Table 3-5. Fundamental Types

Type	ISO C Types	sizeof	Alignment	Description
Boolean	_Bool	1	byte	boolean
Character	char unsigned char	1	byte	unsigned byte
	signed char	1	byte	signed byte
	short signed short	2	halfword	signed halfword
	unsigned short	2	halfword	unsigned halfword
Enumeration	signed enum	4	word	signed word
	unsigned enum	4	word	unsigned word
Integral	int signed int long int signed long	4	word	signed word
	unsigned int unsigned long	4	word	unsigned word
	long long signed long long	8	doubleword	signed doubleword
	unsigned long long	8	doubleword	unsigned doubleword
Pointer	any * any (*) ()	4	word	unsigned word
Floating	float	4	word	single-precision float
	double	8	doubleword	double-precision float

A NULL pointer has all bits zero.

Note: A boolean value is represented as a byte with value 0 or 1. If a byte with a value other than 0 or 1 is evaluated as a boolean value (for example, through the use of unions), the behavior is undefined.

Note: If an enumerated type contains a negative value, it is compatible with and has the same representation and alignment as int; otherwise it is compatible with and has the same representation and alignment as unsigned int.

Note: For each real floating-point type there is a corresponding complex type. This has the same alignment as the real type and twice the size; the representation is the real part followed by the imaginary part.

ATR-SPE

Table 3-6. SPE Types

Type	SPEPIM C Types	sizeof	Alignment	Description
vector-64	__ev64_u16__	8	doubleword	vector of four unsigned halfwords
	__ev64_s16__	8	doubleword	vector of four signed halfwords
	__ev64_u32__	8	doubleword	vector of two unsigned words
	__ev64_s32__	8	doubleword	vector of two signed words
	__ev64_fs__	8	doubleword	vector of two single-precision floats
	__ev64_u64__	8	doubleword	1 unsigned doubleword
	__ev64_s64__	8	doubleword	1 signed doubleword
	__ev64_opaque__	8	doubleword	any of the above

ATR-LONG-DOUBLE-IS-DOUBLE

Table 3-10. Long Double Is Double Type

Type	ISO C Types	sizeof	Alignment	Description
long double is double	long double	8	doubleword	double-precision float

ATR-LONG-DOUBLE-IS-DOUBLE || ATR-LONG-DOUBLE-IBM

This ABI provides the following choices for implementation of long double in compilers and systems:

ATR-LONG-DOUBLE-IS-DOUBLE

- Do not support any floating-point types with greater precision than double. In this case, long doubles and doubles have the same size and precision.

3.1.2.3. Aggregates and Unions

The following are the rules for aggregates (structures and arrays) and unions that apply to their alignment and size.

- The entire aggregate or union must be aligned to its most strictly aligned member, which corresponds to the member with the largest alignment, including flexible array members.
- Each member is assigned the lowest available offset that meets the alignment requirements of the member. Depending on the previous member, internal padding can be required.
- The entire aggregate or union must have a size that is a multiple of its alignment. Depending on the last member, tail padding can be required.

For the following figures, the big-endian byte offsets are located in the upper left corners, and the little-endian byte offsets are located in the upper right corners.

Figure 3-1. Structure Smaller Than a Word

```
struct {
   char c;
};

byte aligned, sizeof is 1
```

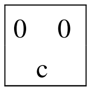
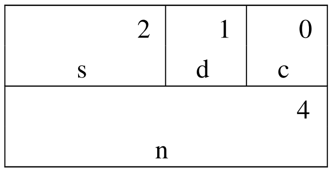


Figure 3-2. Structure With No Padding

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

word-aligned, sizeof is 8
```

little-endian



big-endian

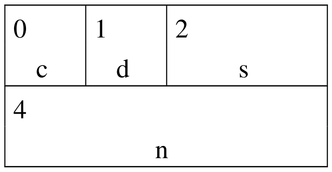
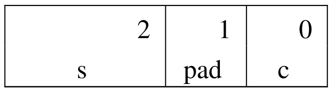


Figure 3-3. Structure With Internal Padding

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

```
halfword-aligned, sizeof is 4
```

little-endian



big-endian

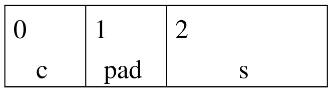
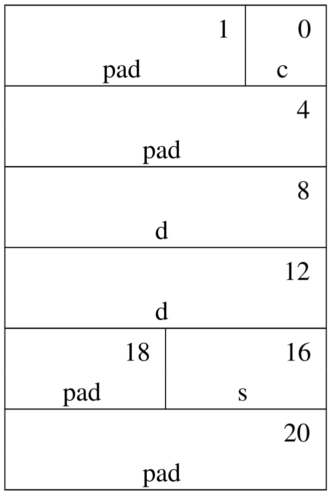


Figure 3-4. Structure With Internal and Tail Padding

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

```
doubleword-aligned, sizeof is 24
```

little-endian



big-endian

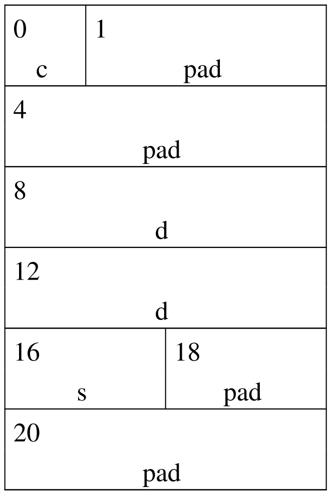
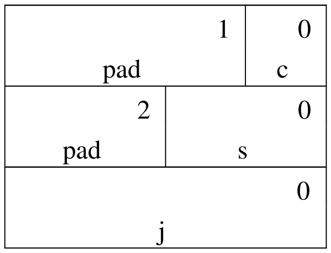


Figure 3-5. Union Allocation

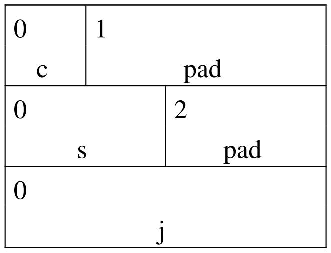
```
union {
  char  c;
  short s;
  int   j;
};

word-aligned, sizeof is 4
```

little-endian



big-endian



3.1.2.4. Bit-fields

Bit-fields can be present in definitions of C structures and unions. These bit-fields define whole objects within the structure or union where the number of bits in the bit-field is specified.

In the following table, a signed range goes from $-(2^{(w-1)})$ to $(2^{(w-1)}) - 1$ and an unsigned range goes from 0 to $(2^{w}) - 1$.

Table 3-11. Bit-Field Types

Bit-field Type	Width (w)
_Bool	1
signed char unsigned char	1 to 8
signed short unsigned short	1 to 16
signed int signed long unsigned int unsigned long enum	1 to 32
signed long long unsigned long long	1 to 64

Bit-fields can be signed or unsigned of type short, int, long, or long long. However, bit-fields shall have the same range for each corresponding type; for example, signed short must have the same range as unsigned short. All members of structures and unions must comply with the size and alignment rules including bit-fields. The following list of size and alignment rules additionally apply to bit-fields:

- The allocation of bit-fields is determined by the system endianess. For little-endian implementations the bit allocation is from the least significant (right) end to the most significant (left) end. The reverse is true for big-endian implementations; the bit allocation is from most significant (left) end to the least significant (right) end.
- A bit-field cannot cross its unit boundary; it must occupy the storage unit allocated for its declared type.
- If there is enough space within a storage unit, bit-fields must share the storage unit with other structure members, including members that are not bit-fields. Clearly all the structure members occupy different parts of the storage unit.
- The types of unnamed bit-fields have no effect on the alignment of a structure or union. However the offsets of an individual bit-field's member must comply with the alignment rules. An unnamed bit-field of zero width causes sufficient padding (possibly none) to be inserted for the next member, or the end of the structure if there are no more nonzero width members, to have an offset from the start of the structure that is a multiple of the size of the declared type of the zero-width member.

The byte offsets for structure and union members are shown in the examples below. The little-endian byte offsets are given in the upper right corners, and the big-endian byte offsets are given in the upper left corners. The bit numbers are given in the lower corners.

Table 3-12. Bit Numbering for 0x01020304

0 3	1 2	2 1	3 0
01	02	03	04
0 7	8 15	16 23	24 31

Figure 3-6. Simple Bit-field Allocation

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

word-aligned, sizeof is 4
```

little-endian

			0
pad	m	k	j
0 13	14 20	21 26	27 31

big-endian

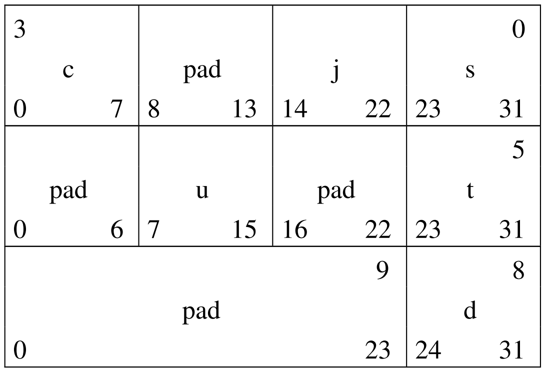
0			
j	k	m	pad
0 4	5 10	11 17	18 31

Figure 3-7. Bit-Field Allocation With Boundary Alignment

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

word-aligned, sizeof is 12
```

little-endian



big-endian

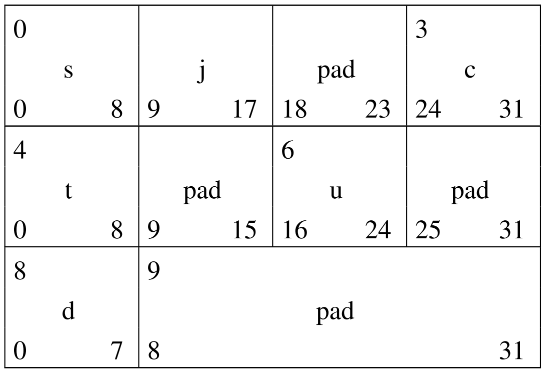
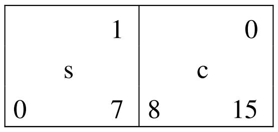


Figure 3-8. Bit-Field Allocation With Storage Unit Sharing

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

halfword-aligned, sizeof is 2
```

little-endian



big-endian

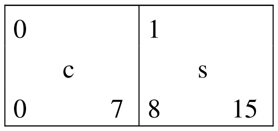


Figure 3-9. Bit-Field Allocation In A Union

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

halfword-aligned, sizeof is 2
```

little-endian

1	0
pad	c
0 7	8 15
1	0
pad	s
0 7	8 15

big-endian

0	1
c	pad
0 7	8 15
0	1
s	pad
0 7	8 15

Figure 3-10. Bit-Field Allocation With Unnamed Bit-Fields

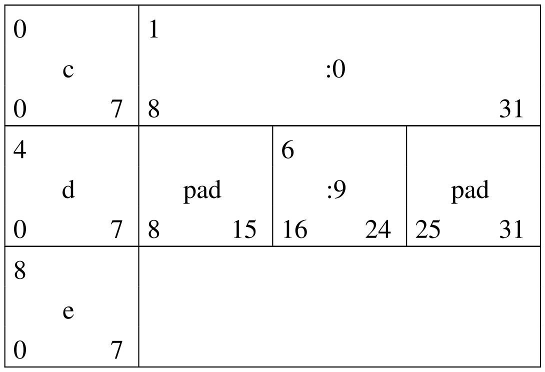
```
struct {
  char  c;
  int   : 0;
  char  d;
  short : 9;
  char  e;
};

byte aligned, sizeof is 9
```

little-endian

		1	0
	:0		c
0		23	24 31
	6		4
pad	:9	pad	d
0 6	7 15	16 23	24 31
			8
			e
			24 31

big-endian



Note: In *Figure 3-10* the alignment of the structure is not affected by the unnamed short and int fields. The named members are aligned relative to the start of the structure. However, it is possible that the alignment of the named members is not on optimum boundaries in memory. For instance, in an array of the structure in *Figure 3-10*, the *d* members will not all be on 4-byte (integer) boundaries.

3.2. Function Calling Sequence

The standard sequence for function calls is outlined in this section. The layout of the stack frame, the parameter passing convention, and the register usage is also detailed in this section. Standard library functions use these conventions, except as documented for the register save and restore functions.

The conventions given in this chapter are adhered to by C programs. Further information on the implementation of C is given in *Section 3.3*.

Note: While it is recommended that all functions use the standard calling sequence, the requirements of the standard calling sequence are only applicable to global functions. Different calling sequences and conventions can be employed by local functions which cannot be reached from other compilation units, if they comply with the stack back trace requirements.

ATR-LONG-DOUBLE-IS-DOUBLE

Note: If long double has the same representation as double, then all statements about how double values are passed to and returned from functions also apply to long double, and all statements about how _Complex double values are passed to and returned from functions also apply to _Complex long double.

ATR-PASS-COMPLEX-AS-STRUCT

Note: For the purposes of the function calling sequence, the C99 _Complex types are treated as if they were represented as a structure containing an array of size two of the corresponding floating point types. That is, a _Complex float is passed to a function and returned from a function as if it were represented as:

```
struct
  {
    float real[2];
  };
```

3.2.1. Registers

Programs and compilers may freely use all registers except those reserved for system use. The system signal handlers are responsible for preserving the original values upon return to the original execution path. Signals that can interrupt the original execution path are documented in (BA-OS) in the System V Interface Definition.

The tables in *Section 3.2.1.1* give an overview of the registers that are global during program execution. The tables use three terms to describe register *Preservation Rules*:

nonvolatile

A *caller* can expect that the contents of all registers marked *nonvolatile* are valid after control returns from a function call.

A *callee* shall save the contents of all registers marked *nonvolatile* prior to modification. The callee must restore the contents of all such registers before returning to its caller.

volatile

A *caller* cannot trust that the contents of registers marked *volatile* have been preserved across a function call.

A *callee* need not save the contents of registers marked *volatile* before modification.

limited-access

The contents of registers marked *limited-access* have special preservation rules. These registers have mutability restricted to certain bit-fields as defined by the Power ISA. The individual bits of these bit-fields are defined by this ABI to be *limited-access*.

Under normal conditions a *caller* can expect that these bits have been preserved across a function call. Under the special conditions, indicated in *Section 3.2.1.2*, a *caller shall expect* that these bit will have changed across function calls even if they have not.

A *callee* may only permanently modify these bits without preserving the state upon entrance to the function if the *callee* satisfies the special conditions indicated in *Section 3.2.1.2*; otherwise, these bits must be preserved before modification and restored before returning to the caller.

3.2.1.1. Register Roles

In the 32-bit Power Architecture, there are always 32 general-purpose registers, each 32 bits wide. Throughout this document the symbol r*N* is used, where *N* is a register number, to refer to general-purpose register *N*.

Table 3-13. Register Roles

Register	Preservation Rules	Purpose
r0	volatile	Optional in function linkage
r1	nonvolatile	Stack frame pointer
r2	nonvolatile	See the following table
r3-r6	volatile	Parameter and return value
r7-r10	volatile	Additional function parameters
r11-r12	volatile	Optional in function linkage
r13	nonvolatile	Small data area pointer
r14-r31	nonvolatile	Local variables
LR	volatile	Link register
CTR	volatile	Loop count register
XER	volatile	Fixed point exception register
CR0-CR1	volatile	Condition register fields
CR2-CR4	nonvolatile	Condition register fields
CR5-CR7	volatile	Condition register fields

Optional Function Linkage

A function cannot depend on the values of those registers optional in the function linkage (r0, r11, and r12) because they may be altered by inter-library calls.

Stack Frame Pointer

The stack pointer always points to the lowest allocated valid stack frame. It must maintain quadword alignment and grow toward the lower addresses. The contents of the word at that address always points to the previously allocated stack frame. A called function is permitted to decrement it if required. See *Section 3.3.9* for additional information.

Small Data Area Pointer

Register r13 is the small data area pointer. Process start up code for executables that reference data in the small data area with 16-bit offset addressing relative to r13 must load the base of the small data area (the value of the dynamic linker-defined symbol `_SDA_BASE_`) into r13. Shared objects shall not alter the value in r13. See *Section 4.8* for more details.

Link Register

The link register contains the address a called function normally returns to. It is volatile across function calls.

Condition Register Fields

In the condition register, the bit-fields CR2, CR3, and CR4 are nonvolatile and the value on entry must be restored on exit. The other bit-fields are volatile. The bit-field CR6 shall be set by the caller of a variable argument list function as described in *Section 3.2.4*.

Register r2 shall contain the base of the small data area 2 (the value of the dynamic linker-defined symbol `_SDA2_BASE_`) which is used for referencing the ELF sections named .PPC.EMB.sdata2 and .PPC.EMB.sbss2, if either section exists in an executable. The small data area 2 base is an address such that every byte in the two sections is within a signed 16-bit offset of that address, which is analogous to the use of r13, as described previously, to contain `_SDA_BASE_`, which is the base of sections .sdata and .sbss. A routine in a shared object shall not use r2. See *Section 4.8.2* for more details.

Table 3-15. EABI Register Role for General-Purpose Register 2

Register	Preservation Rules	Purpose
r2	nonvolatile	SDA2 (Small Data Area 2) pointer.

ATR-PASS-COMPLEX-IN-GPRS

Table 3-16. Register Roles for the _Complex float and _Complex double Types

Register	Preservation Rules	Purpose
r3-r10	volatile	Used for _Complex float and _Complex double *parameters and return values.*

ATR-CLASSIC-FLOAT

On Power Architecture processors that support Power ISA category *Floating-point*, there are always 32 floating-point registers, each 64 bits wide, and an associated special-purpose register to provide floating-point status and control. Throughout this document the symbol f*N* is used, where *N* is a register number, to refer to floating-point register *N*.

Table 3-19. Floating-Point Register Roles for Binary Floating-Point Types

Register	Preservation Rules	Purpose
f0	volatile	
f1	volatile	Used for *parameter passing* and *return values* of binary float types.
f2-f8	volatile	Used for *parameter passing* of binary float types.
f9-f13	volatile	
f14-f31	nonvolatile	
FPSCR	limited-access	Floating point status and control register limited-access bits. Preservation rules governing the limited-access bits for the bit-fields [VE], [OE], [UE], [ZE], [XE], and [RN] are presented in *Section 3.2.1.2*.

ATR-SOFT-FLOAT

Table 3-21. Soft-Float General-Purpose Register Roles for Binary Floating-Point Types

Register	Preservation Rules	Purpose
r3-r10	volatile	Volatile parameter and return value registers for float, double, and long double binary floating-point types. If the parameters are within the first eight words of the parameter list: • Float values occupy a single GPR. • Double values occupy adjacent GPRs. • Long double values occupy four adjacent GPRs. There are special rules governing how parameters that span multiple GPRs should be split between registers and the parameter save area outlined in *Section 3.2.3*.

ATR-SPE

The ISA Signal Processing Engine (SPE) category provides upper words for the 32 general-purpose registers, thus allowing them to be used in SPE APU operations to hold two 32-bit words. The Signal Processing Engine category also provides several special-purpose registers. The volatility of all 64-bit registers is the same for the upper and lower word. If only the lower word is modified by a function, only the lower word need be saved and restored.

Table 3-24. SPE Register Roles

Register	Preservation Rules	Purpose
SPEFSCR	limited-access	Signal processing and embedded floating-point status and control register. Preservation rules governing the limited-access bits for the bit-fields [FINXE], [FINVE], [FDBZE], [FUNFE], [FOVFE], and [FRMC] are presented in *Section 3.2.1.2*.
ACC	volatile	64-bit SPE accumulator register.

3.2.1.2. Limited-Access Bits

The Power ISA identifies a number of registers which have mutability limited to the specific bit-fields indicated in the following list:

ATR-CLASSIC-FLOAT

FPSCR [VE]

The *Floating-Point Invalid Operation Exception Enable* bit [VE] of the FPSCR register.

ATR-CLASSIC-FLOAT

FPSCR [OE]

The *Floating-Point Overflow Exception Enable* bit [OE] of the FPSCR register.

ATR-CLASSIC-FLOAT

FPSCR [UE]

The *Floating-Point Underflow Exception Enable* bit [UE] of the FPSCR register.

ATR-CLASSIC-FLOAT

FPSCR [ZE]

The *Floating-Point Zero Divide Exception Enable* bit [ZE] of the FPSCR register.

ATR-CLASSIC-FLOAT

FPSCR [XE]

The *Floating-Point Inexact Exception Enable* bit [XE] of the FPSCR register.

ATR-CLASSIC-FLOAT

FPSCR [RN]

The *Binary Floating-Point Rounding Control* field [RN] of the FPSCR register.

ATR-SPE

SPEFSCR [FINXE]

The *Embedded Floating-Point Round (Inexact) Exception Enable* field [FINXE] of the SPEFSCR register.

ATR-SPE

SPEFSCR [FINVE]

The *Embedded Floating-Point Invalid Operation/Input Error Exception Enable* field [FINVE] of the SPEFSCR register.

ATR-SPE

SPEFSCR [FDBZE]

The *Embedded Floating-Point Divide By Zero Exception Enable* field [FDBZE] of the SPEFSCR register.

ATR-SPE

SPEFSCR [FUNFE]

The *Embedded Floating-Point Underflow Exception Enable* field [FUNFE] of the SPEFSCR register.

ATR-SPE

SPEFSCR [FOVFE]

The *Embedded Floating-Point Overflow Exception Enable* field [FOVFE] of the SPEFSCR register.

ATR-SPE

SPEFSCR [FRMC]

The *Embedded Floating-Point Rounding Mode Control* field [FRMC] of the SPEFSCR register.

The bits composing these bit-fields are identified as *limited-access* because this ABI manages how they are to be modified and preserved across function calls.

Limited-access bits may be changed across function calls only if the called function has specific permission to do so as indicated by the following conditions.

A function without permission to change the *limited-access* bits across a function call shall save the value of the register before modifying the bits and restore it before returning to its calling function.

Limited-Access Conditions

- Standard library functions expressly defined to change the state of limited-access bits are not constrained by nonvolatile preservation rules, e.g., the `fesetround()` and `feenableexcept()` functions.
- All other standard library functions shall save the old value of these bits on entry, change the bits for their purpose, and restore the bits before returning.
- Where a standard library function such as `qsort()` calls functions provided by an application the following rules shall be observed:
    - The limited-access bits on entry to the first call to such a callback must have the values they had on entry to the library function.
    - The limited-access bits on entry to a subsequent call to such a callback must have the values they had on exit from the previous call to such a callback.
    - The limited-access bits on exit from the library function must have the values they had on exit from the last call to such a callback.
- The compiler can directly generate code that saves and restores the limited-access bits.
- The values of the limited-access bits are unspecified on entry into a signal handler because a library or user function can temporarily modify the limited-access bits when the signal was taken.

- When `setjmp()` returns from a direct invocation, the limited-access bits must have the values they had on entry to setjmp; when it returns from a call to `longjmp()`, the limited-access bits must have the values they had on entry to `longjmp()`.

ATR-CLASSIC-FLOAT

- C Library intrinsics, such as `_FPU_SETCW()`, may modify the limited-access bits of the FPSCR.

ATR-SPE

- The following intrinsics defined by the SPE PIM may change the limited-access bits of the SPEFCSR register:

  ```
  __ev_clr_spefscr_sovh() __ev_clr_spefscr_sov() __ev_clr_spefscr_finxs()
  __ev_clr_spefscr_finvs() __ev_clr_spefscr_fdbzs() __ev_clr_spefscr_funfs()
  __ev_clr_spefscr_fovfs() __ev_set_spefscr_frmc()
  ```

ATR-SOFT-FLOAT

- Any data stored internally by software floating-point code to describe rounding modes and enabled exceptions is subject to the same rules as limited-access register bits.

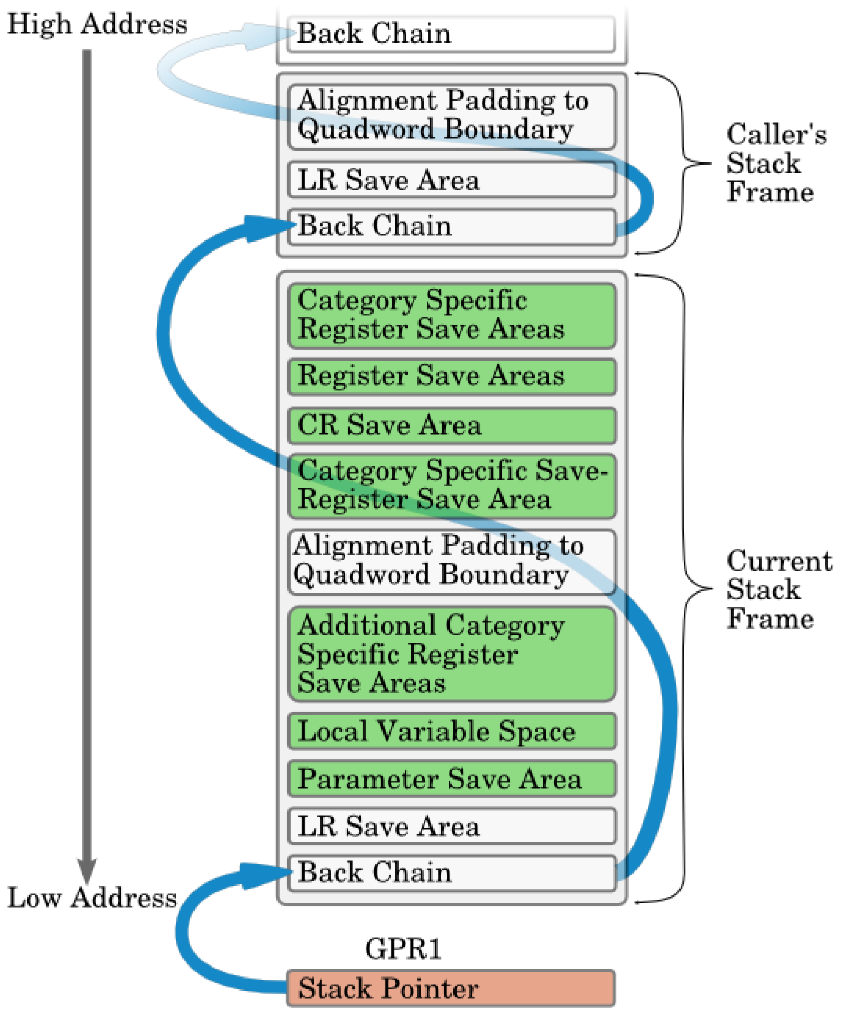
Note: The unwinder does not need to make specific allowances for limited-access bits.

3.2.2. The Stack Frame

A function shall establish a stack frame if it requires the use of nonvolatile registers, its local variable usage can't be optimized into registers, or it calls another function. It need only allocate space for the required stack frame elements, namely the *backchain pointer*, the *LR save area*, and *padding* to the required alignment.

Figure 3-11 shows the relative layout of an allocated stack frame following a nonleaf function call, where the *stack pointer* points to the *backchain* word of the caller's stack frame. In general the *stack pointer* always points to the *backchain* word of the most recently allocated stack frame.

Figure 3-11. Stack Frame Organization



In Figure 3-11 the green areas indicate an *optional* save area of the stack frame. Refer to *Section 3.2.2.2* for a description of the optional save areas described by this ABI.

3.2.2.1. General Stack Frame Requirements

The following general requirements apply to all stack frames:

- The stack shall be quadword-aligned.
- The minimum stack frame size shall be 16 bytes. A minimum stack frame consists of the first two words (*backchain* word and *LR save word*), with padding to meet the 16-byte alignment requirement.
- There is no maximum stack frame defined.
- Padding shall be added to the *local variable space* of the stack frame to maintain the defined stack frame alignment in the absence of register save areas.

- The *stack pointer* (r1), shall always point to the lowest address word of the most recently allocated stack frame.
- The stack shall start at high addresses and grow downward toward lower addresses.
- The lowest address word (the *backchain* word in Figure 3-11) shall point to the previously allocated stack frame. An exception occurs with the first stack frame, which shall have a value of 0 (NULL).
- If required, the stack pointer shall be decremented in the called function's prologue and restored in the called function's epilogue.
- The *stack pointer shall be updated atomically* so that, at all times, it points to a valid *backchain* word. This update may be achieved in a number of ways, as indicated in *Section 3.3.3.3*.
- Before a function calls any other functions, it shall save the value of the LR register into the *LR save area* of the caller's stack frame.

> Note: An optional frame pointer may be created if necessary (e.g., as a result of dynamic allocation on the stack as described in *Section 3.3.9*) to address arguments or local variables.

A sample of a minimum stack frame allocation is demonstrated in *Figure 3-12* containing these requirements.

Figure 3-12. Example Minimum Stack Frame Allocation

```
stwu 1,-32(1)       - Store backchain, decr SP
mflr 0              - Copy LR to R0
stw  0,36(1)        - Store LR in previous LR save area
```

3.2.2.2. Optional Save Areas

This ABI provides a stack frame with a number of optional save areas. This section will indicate the relative position of these save areas in relation to each other and the primary elements of the stack frame.

Because the back chain word of a stack frame must maintain quadword alignment the following save area diagrams indicate that an optional *special purpose padding* element might be necessary near the low-address end of a stack frame (above the link register save).

An optional *alignment padding to quadword boundary* element might be necessary near the high-address end of the stack in order to quadword-align the low-address beginning of a register save area immediately below it, e.g, *Figure 3-18*.

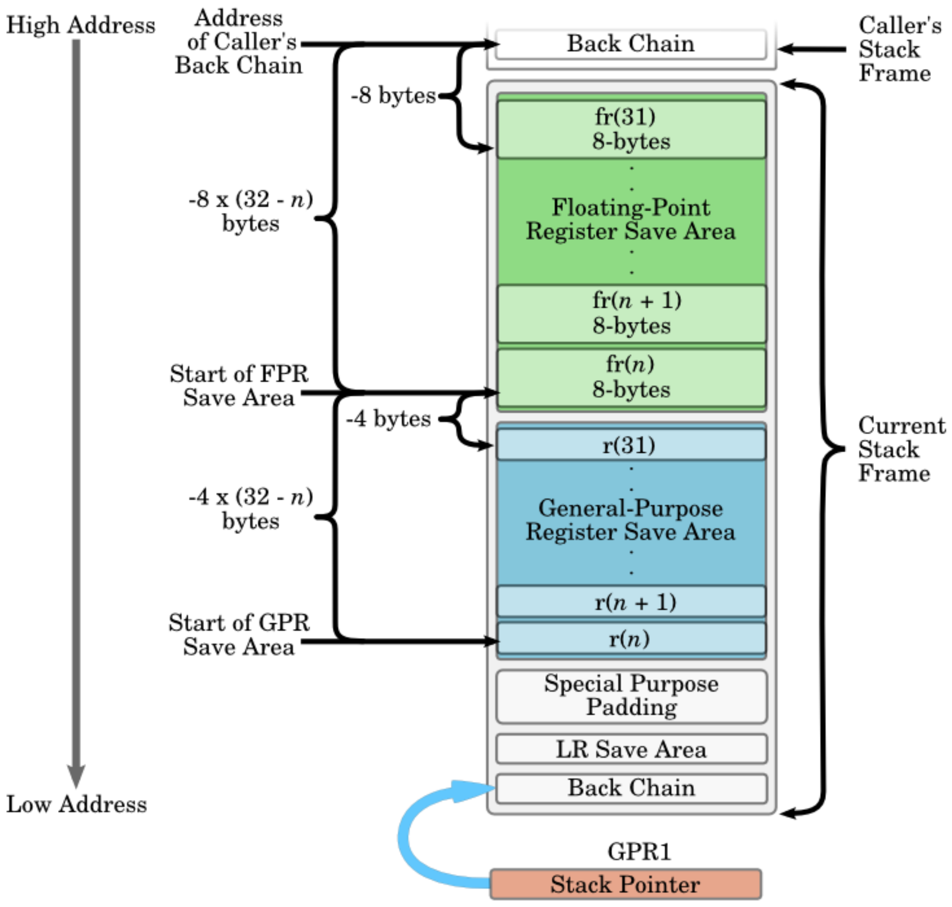
Register Save Areas

ATR-CLASSIC-FLOAT

Floating-Point Register Save Area

If a function is to change the value in any nonvolatile floating-point register fr*n* it shall first save the value fr*n* in the *Floating-Point Register Save Area* in a doubleword located 8 × (32 - *n*) bytes before the back chain word of the previous frame, as shown in *Figure 3-13*.

Figure 3-13. General-Purpose and Floating-Point Register Save Areas



ATR-CLASSIC-FLOAT

General-Purpose Register Save Area (with floating-point registers available)

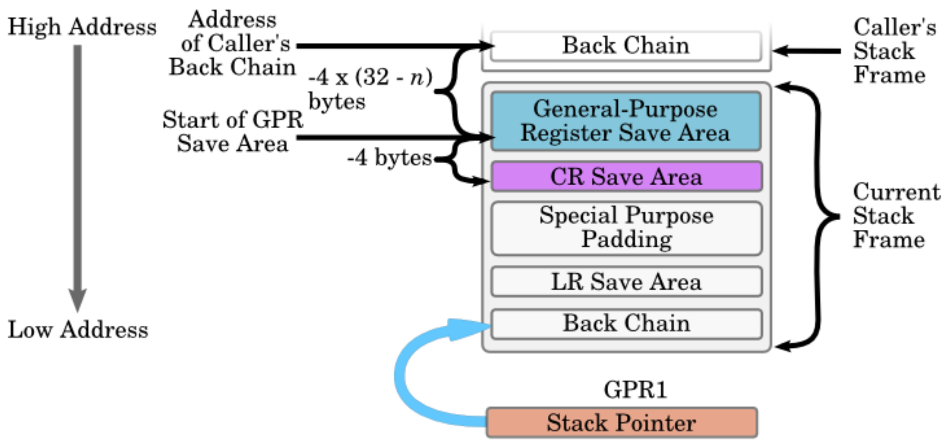
If a function is to change the value in any nonvolatile general-purpose register r*n*, it shall first save the value of r*n* in the *general register save area*, in a word located 4 × (32 - *n*) bytes before the low-addressed end of the *Floating-Point Register Save Area*, as shown in *Figure 3-13*.

CR Save Area

CR Save-Register Save Area

If a function changes the value in any nonvolatile field of the condition register, it shall first save the value in all the nonvolatile fields of the condition register in the *CR Save Area*, which is the word below the low address end of the *general register save area*, as shown in Figure 3-15.

Figure 3-15. CR Save Area



ATR-CLASSIC-FLOAT

Figure 3-16. CR Save Area With Floating-Point Save Area

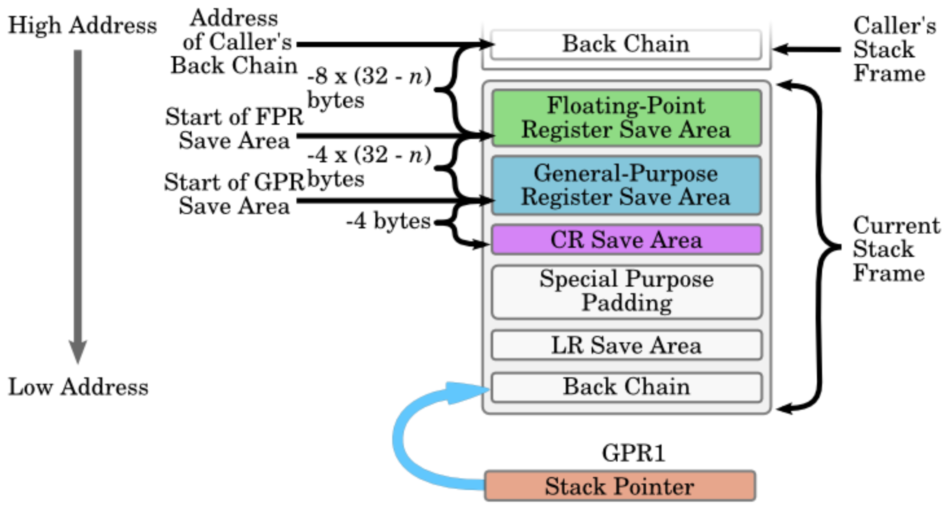


Figure 3-16 shows the location of the CR save area when a floating-point save area is present.

Category Specific Save-Register Save Area

!ATR-VECTOR

The category-specific save-register save area is unnecessary.

Category-Specific Register Save Areas

!ATR-VECTOR

The section *Category-Specific Register Save Areas* has no defined elements.

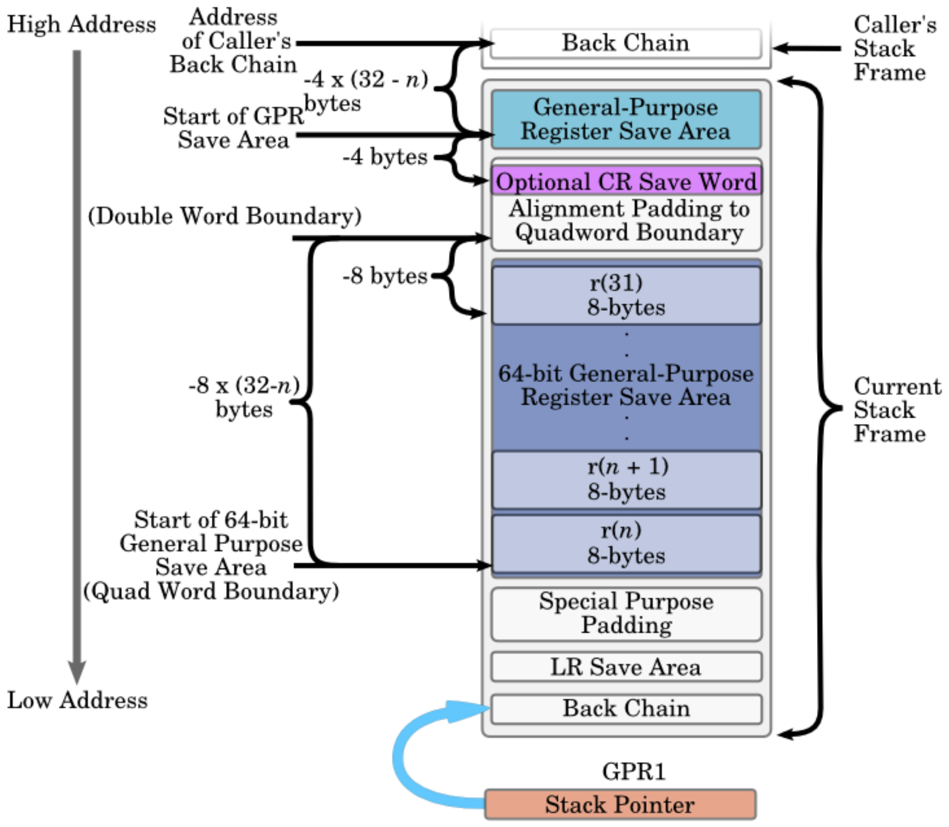
Additional Category Specific Register Save Areas

ATR-SPE

SPE 64-bit General-Purpose Register Save Area

If a function changes the value in the upper word of any nonvolatile general-purpose register r*n*, it shall first save the value of r*n* in the *64-bit general-purpose register save area*, in a doubleword located 8 × (32 - *n*) bytes before the low-addressed end of the *CR save area* (plus any required padding) if the *CR Save Area* is present. Otherwise, it is located in a doubleword 8 × (32 - *n*) bytes before the low-address end of the *General-Purpose Register Save Area* (plus any required padding). The *64-bit General-Purpose Save Area* shall have quadword alignment. While not technically necessary, quadword alignment is required for congruence with AltiVec and VMX technology.

Figure 3-18. SPE 64-bit General-Purpose Register Save Area

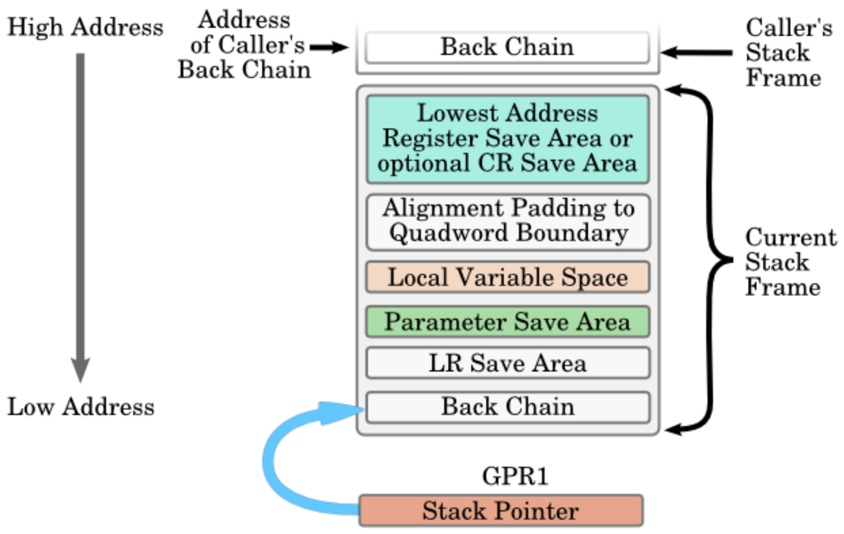


Note: The purpose of providing both 32-bit and 64-bit general register save areas is to reduce the stack usage for routines that use only the lower word of some nonvolatile registers, and both the lower and upper word of some other nonvolatile registers. A compiler may choose to save and restore all 64 bits of each modified nonvolatile general-purpose register, as long as the debugging information reflects this choice.

ATR-SPE

Note: If the compiler uses the 32-bit general save areas when possible, routines compiled in this manner that do not use any of the 64-bit instructions in the SPE architecture should remain Power Architecture EABI compliant (both in regards to stack layout, and in all other ways).

Figure 3-19. Parameter Save Area and Local Variable Space



Parameter Save Area

Parameter Save Area

The *Parameter Save Area* shall be allocated by the caller, and shall be large enough to contain the parameters needed by the caller. The calling function cannot expect that the contents of this save area are valid when returning from the callee. Refer to *Figure 3-19* for information on the location of this space.

Local Variable Space

Local Variable Space

The *Local Variable Space* is used for allocation of local variables. If the *Parameter Save Area* is in use, the *Local Variable Space* is located immediately above it, at a higher address, otherwise it is located immediately above the LR Save word. There is no restriction on the size of this area. Refer to *Figure 3-19* for information on the location of this space.

3.2.3. Parameter Passing

For the Power Architecture, it is more efficient to pass arguments to functions in registers, rather than through memory. For the Power Architecture, the following parameters can be passed in registers.

- Up to eight arguments can be passed in general-purpose registers r3 through r10

ATR-SPE

- Up to eight 64-bit doubleword vector arguments are passed in general-purpose registers.

ATR-CLASSIC-FLOAT

- Up to eight floating-point arguments can be passed in floating-point registers f1 through f8.

If fewer arguments are needed, then the unused registers defined previously will contain undefined values on entry to the called function.

If there are more arguments than registers, then a function must provide space for the arguments in its stack frame. When this happens, only the minimum storage needed to contain the extra arguments needs to be allocated in the stack frame.

The following algorithm describes where arguments are passed for the C language. In this algorithm, arguments are assumed to be ordered from left (first argument) to right. The actual order of evaluation for arguments is unspecified.

gr contains the number of the next available general-purpose register.

ATR-CLASSIC-FLOAT

fr contains the number of the next available floating-point register.

3.2.3.1. Parameter Passing Register Selection Algorithm

> Note: The following types refer to the type of the argument as declared by the function prototype. The argument values will be converted (if necessary) to the types of the prototype arguments before passing them to the called function.
>
> If a prototype is not present, or it is a variable argument prototype and the argument is after the ellipsis, the type refers to the type of the data objects being passed to the called function.

- ***INITIALIZE***: If the function return type requires a storage buffer, set *gr* = 4, else set *gr* = 3.

  ATR-CLASSIC-FLOAT

  Set *fr* = 1

  Set *starg* to the address of parameter word 1.

- ***SCAN***: If there are no more arguments, terminate. Otherwise, select one of the following depending on the type of the next argument:
  - **SINGLE_GP**:
    - A single integer no more than 32 bits

ATR-SOFT-FLOAT

- A single-precision floating-point value if prototype is present

ATR-SPE

- A 64-bit vector if the called function is not a variable-argument function

- A pointer to a data object
- A struct or union that shall be treated as a pointer to the data object, or to a copy of the data object when necessary to enforce call-by-value semantics. Only if the caller can ascertain that the data object is constant can it pass a pointer to the data object itself.

  ATR-PASS-COMPLEX-AS-STRUCT

  This pointer treatment includes complex single-precision, double-precision, and quad-precision floating-point values.

If *gr* > 10, go to *OTHER*. Otherwise, load the argument value into general-purpose register *gr*, set $gr = gr + 1$, and go to *SCAN*. Values shorter than 32 bits are sign-extended or zero-extended, depending on whether they are signed or unsigned.

- **DUAL_GP**:
  - A 64-bit integer

ATR-SOFT-FLOAT

- A double-precision floating-point value

ATR-SPE

- A 64-bit vector being passed to a variable-argument function

ATR-PASS-COMPLEX-IN-GPRS

- A complex single-precision float

If *gr* $>$ 9, go to *OTHER*. If *gr* is even, set *gr* = *gr* + 1. Load the lower-addressed word of the argument into *gr* and the higher-addressed word into *gr* + 1, set *gr* = *gr* + 2, and go to *SCAN*.

- **QUAD_GP**:

ATR-PASS-COMPLEX-IN-GPRS

- A complex double-precision float

If *gr* $>$ 7, go to *OTHER*. Load the words of the argument, in memory-address order, into *gr*, *gr* + 1, *gr* + 2 and gr + 3, set *gr* = *gr* + 4, and go to *SCAN*.

ATR-CLASSIC-FLOAT

- **SINGLE_FP**:
  - A single-precision floating-point value or a double-precision floating-point value

  if *fr* $>$ 8, go to *OTHER*. Otherwise load the argument into register *fr*, set *fr* to *fr* + 1, and go to *SCAN*

- ***OTHER***:
  - Arguments not otherwise handled are passed in the parameter save area of the caller's stack frame. Most of the types handled in SINGLE_GP, as defined previously, are considered to have 4-byte size and alignment, with simple integer types shorter than 32 bits sign- or zero-extended to 32 bits. Long long arguments are considered to have 8-byte size and alignment. The same 8-byte arguments that must go in aligned pairs or registers are 8-byte aligned on the stack.

  ATR-PASS-COMPLEX-IN-GPRS

  Complex single-precision float arguments are considered to have 8-byte size and alignment.

  ATR-SPE

  64-bit vector arguments are considered to have 8-byte size and alignment.

Round *starg* up to a multiple of the alignment requirement of the argument and copy the argument byte-for-byte, beginning with its lowest addressed byte, into *starg*, ..., *starg* + *size* - 1. Set *starg* to *starg* + *size*, and go to *SCAN*.

Types handled in QUAD_GP, as defined previously, are only 4-byte aligned when passed on the stack.

If *gr* > 9 and the type is DUAL_GP, or *gr* > 7 and the type is QUAD_GP, then set *gr* = 11 (to prevent subsequent SINGLE_GPs from being placed in registers after DUAL_GP, QUAD_GP, or EIGHT_GP arguments that would no longer fit in the registers).

3.2.3.2. Parameter Passing Examples

The following section provides some examples using the algorithm described in *Section 3.2.3.1*.

ATR-CLASSIC-FLOAT && ATR-LONG-DOUBLE-IBM || ATR-LONG-DOUBLE-IS-DOUBLE

Figure 3-20. Parameter Passing Example

```
typedef struct {
  int    a;
  double dd;
} sparm;
sparm   s, t;
int     c, d, e;
long double ld;
double  ff, gg, hh;

x = func(c, ff, d, ld, s, gg, t, e, hh);
```

ATR-CLASSIC-FLOAT && ATR-LONG-DOUBLE-IS-DOUBLE

Table 3-27. Parameter Passing Using long double is double

Parameter	Register	Byte Offset In Parameter Save Area
c	r3	(not stored in parameter save area)
d	r4	(not stored)
ld	f1	(not stored)
ptr to s	r5	(not stored)
ff	f2	(not stored)
gg	f3	(not stored)
ptr to t	r6	(not stored)
e	r7	(not stored)
hh	f4	(not stored)

ATR-SOFT-FLOAT && ATR-LONG-DOUBLE-IS-DOUBLE

Table 3-28. Parameter Passing Using long double is double and Soft-Float

Parameter	Register	Byte Offset In Parameter Save Area
c	r3	(not stored in parameter save area)
ff	r5,r6	(not stored)
d	r7	(not stored)
ld	r9,r10	(not stored)
ptr to s	(none)	08-11 (stored in parameter save area)
gg	(none)	16-23 (stored)
ptr to t	(none)	24-27 (stored)
e	(none)	28-31 (stored)
hh	(none)	32-39 (stored)

ATR-SPE

Figure 3-22. SPE Parameter Passing Example

```
typedef struct {
  int     a;
  double dd;
} sparm;
sparm    s;
int      c;
__ev64_opaque__ va, vb;
```

```
float ff;
double gg;
x = func(c, ff, va, gg, vb, s);
```

ATR-SPE

Table 3-30. Parameter Passing of SPE Data Types

Parameter	Register	Byte Offset In Parameter Save Area
c	r3	(not stored in parameter save area)
ff	r4	(not stored)
va	r5	(not stored)
gg	r7, r8	(not stored)
vb	r9	(not stored)
ptr to s	r10	(not stored)

3.2.4. Variable Argument Lists

C programs that are intended to be portable across different compilers and architectures must use the header file *<stdarg.h>* to deal with variable argument lists. This header file contains a set of macro definitions that define how to step through an argument list. The implementation of this header file may vary across different architectures, but the interface is the same.

C programs that do not use this variable argument list header file, and assume that all the arguments are passed on the stack in increasing order on the stack are not portable, especially on architectures that pass some of the arguments in registers. The Power Architecture is one of the architectures that passes some of the arguments in registers.

ATR-CLASSIC-FLOAT

CR bit 6 must be set by a variable argument list function caller that passes any arguments in floating-point registers. The recommended instruction to achieve this is: `creqv 6,6,6`. It is recommended that CR bit 6 be cleared by variable argument list function callers that do not pass any arguments in floating-point registers, using the instruction `crxor 6,6,6`.

The parameter list may be zero length and is only allocated when parameters are spilled.

ATR-SPE

For variable argument functions, 64-bit vectors (both before and after the ellipsis) are passed in the low words of two consecutive registers, in the same manner as long long variables.

3.2.5. Return Values

ATR-CLASSIC-FLOAT

Functions that return float or double values shall place the result in register f1. The float values will be rounded to single-precision.

ATR-SOFT-FLOAT

Functions shall return single-precision float values in r3, and double-precision values shall be returned with the low addressed word in r3 and the higher in r4.

ATR-SPE

Functions shall return values of 64-bit vector types in r3.

Functions that return values of the following list of types shall place the result in register r3 as signed or unsigned integers as appropriate, sign extended or zero extended to 32 bits where necessary:

- char
- enum
- short
- int
- long
- pointer to any type.
- _Bool

Aggregates or unions whose size is less than or equal to eight bytes shall be returned in r3 and r4, as if they were first stored in memory area and then the low-addressed word were loaded in r3 and the high-addressed word were loaded into r4. Bits beyond the last member of the structure or union are not defined.

Functions that return structures or unions which do not conform to the requirements of being returned in registers shall place the results in a storage buffer that has been pre-allocated by the caller. The address of this storage buffer shall be passed as the first argument in register r3 as a hidden argument resulting in *gr* being initialized to 4 as opposed to 3 in the argument passing algorithm in *Section 3.2.3.1*.

Functions that return values of type long long and unsigned long long shall place the result in registers r3 and r4. The lower addressed word shall be placed in register r3, and the higher addressed word shall be in register r4.

ATR-PASS-COMPLEX-IN-GPRS

Functions that return values of type _Complex float shall place the results in registers r3 and r4. The lower addressed word shall be placed in r3; the higher addressed word shall be in register r4.

ATR-PASS-COMPLEX-IN-GPRS

Functions that return values of type _Complex double shall place the results in registers r3 through r6, from lowest to highest addressed words.

3.3. Coding Examples

The following ISO C coding examples are provided as illustrations of how operations may be done, not how they shall be done, for calling functions, accessing static data, and transferring control from one part of a program to another. They are shown as code fragments with simplifications to explain addressing modes, not necessarily show the optimal code sequences or compiler output. The small data area is not used in any of them.

The previous sections explicitly specify what a program, operating system, and processor may and may not assume and are the definitive reference to be used.

In these examples, absolute code and position-independent code are referenced.

When instructions hold absolute addresses, a program must be loaded at a specific virtual address in order to permit the absolute code model to work.

When instructions hold relative addresses, a program can be loaded at various positions in virtual memory and is referred to as position-independent code model.

3.3.1. Code Model Overview

When a process image is created, an executable has fixed addresses.

3.3.3. Function Prologue and Epilogue

A function’s prologue and epilogue is detailed in this section.

3.3.3.1. The Purpose of a Function’s Prologue

- Create a stack frame when required.
- Save any nonvolatile registers that are used by the function.
- Save any limited-access bits that are used by the function, per the rules described earlier.

3.3.3.2. The Purpose of a Function’s Epilogue

- Restore all registers and limited-access bits that were saved by the function’s prologue.
- Restore the last stack frame.
- Return to the caller.

3.3.3.3. Rules for Prologue and Epilogue Sequences

Set function prologue and function epilogue code sequences are not imposed by this ABI. There are several rules that must be adhered to in order to ensure reliable and consistent call chain backtracing.

- Before a function calls any other function, it shall establish its own stack frame, whose size shall be a multiple of 16 bytes, and shall save the link register at the time of entry in the LR save area of its caller’s stack frame.
- The calling sequence does not restrict how languages leverage the *local variable space* of the stack frame, and there is no restriction on the size of this section.
- The *parameter save area* shall be allocated by the caller, and shall be large enough to contain the parameters needed by the caller. Its contents are not saved across function calls.
- In instances where a function’s prologue creates a stack frame, the backchain word of the stack frame shall be updated atomically with the value of the stack pointer (r1). This task can be done by using one of the following *Store Word with Update* instructions:
  - *Store Word with Update* instruction with relevant negative displacement for stack frames that are smaller than 32 KB.
  - *Store Word with Update Indexed* instruction where the two’s complement size of the stack frame has been computed, using `addis` and `addi` or `ori` instructions, and then loaded into a volatile register for stack frames that are 32 KB or greater.

- The deallocation of a function’s stack frame must be an atomic operation. This task can be accomplished by one of the following methods given below:
  - Increment the stack pointer by the identical value that it was originally decremented in the prologue when the stack frame was created.

  - Load the stack pointer (r1) with the value in the backchain word in the stack frame.

- If any nonvolatile registers are to be used by the function the contents of the register must be saved into a *register save area*. See *Section 3.2.2.2* for information on all of the optional register save areas.

  Saving and/or restoring nonvolatile registers used by the function can be accomplished using in-line code. Alternatively one of the system subroutines described in *Section 3.3.4* may offer a more efficient alternative to in-line code, especially in cases where there are many registers to be saved or restored.

Unlike some other processors that implement the Power Architecture embedded processors may support *load and store multiple* Power Architecture instructions in little-endian mode. On big-endian implementations they may or may not be slower than the register-at-a-time saves, but reduce the instruction footprint.

Position independent functions which make external data references will need to load a nonvolatile register with a pointer to a *Global Offset Table* as show in Figure 3-26. In cases where external data references are only made from within conditional code the loading of a *Global Offset Table* pointer can be delayed until it is needed.

3.3.4. Register Saving and Restoring Functions

This section describes functions that can be used to save and restore contents of nonvolatile registers. The use of these routines, rather than performing these saves and restores inline in the prologue and epilogue of functions, can help reduce code footprint.

This section details register saving and restoring functions. The calling conventions of these functions are not standard and the executables or shared objects that use these functions must statically link them. The specific calling convention for each of these functions is described in *Section 6.1.2*.

ATR-SPE && ATR-SOFT-FLOAT

The use of a merged register file removes the need for distinct routines for saving and restoring floating-point registers. However, in order to conserve stack space, this ABI describes several new routines to allow the compiler to use the minimum stack space for holding copies of nonvolatile registers. See *Section 3.3.4.1* for information on the routines.

ATR-SPE

For situations where stack space is not at a premium, the compiler can elect to only use the 64-bit save and restore functions for functions that require some use of the upper halves of the registers, and traditional 32-bit save and restore functions for code that uses only classic instructions.

There are several cases to consider with respect to saving/restoring nonvolatile registers for a function:

- No nonvolatile registers need saving or restoring.
- Only 32-bit nonvolatile registers need to be saved or restored. In this case, the classic (32-bit) save and restore functions, or the `stmw` and `lmw` instructions, can be used.

ATR-SPE

- Only 64-bit nonvolatile registers need to be saved or restored. In this case, 64-bit versions of the classic save and restore functions can be used. There is no equivalent to `stmw`/`lmw` for both halves of a 64-bit register.
- A mixture of 32-bit and 64-bit nonvolatile registers need saving or restoring. To minimize complexity, the 32-bit nonvolatile registers shall be contiguous and at the upper end of the registers (r*N* - r31). This also allows the `stmw` and `lmw` instructions to still be used, if desired. The 64-bit nonvolatile registers shall also be contiguous (r*M* - r(*N* - 1)). The registers are saved or restored by calling both a 32-bit save and restore function and a 64-bit save and restore function.

Saving and restoring functions also have variants (*_g* for register save routines, *_x* and *_t* for register restore routines) that bundle some common prologue and epilogue operations to reduce overhead and code footprint by a few instructions. These are described in more detail in the following paragraphs.

The 32-bit save and restore functions restore consecutive 32-bit registers from register *m* through register 31.

ATR-SPE

The simple 64-bit save and restore functions restore consecutive 64-bit registers from register *m* through register 31. The more complex (CTR-based) 64-bit save and restore functions save and restore consecutive 64-bit registers from register *m* through register n, and use the value $N - m + 1$ in the CTR register to determine how many registers to save.

Higher-numbered registers are saved at higher addresses within a save area.

All of the 32-bit save and restore functions in this section expect the address of the backchain word to be contained in r11. The back chain word is the next word after the end of the 32-bit general register save area. r11 is not modified by these functions.

ATR-SPE

The value held in r11 for the 64-bit save and restore functions varies on the type of function.

- All the non-CTR 64-bit save and restore functions described in this section expect r11 to contain the address of the backchain word, adjusted by subtracting 144. The adjustment by 144 allows the immediate form of the 64-bit load/store instructions to be used (they have an unsigned immediate).
- The CTR-based 64-bit save and restore functions described in this section expect the CTR to contain the number of registers to save (1:18). Register r11 should be calculated by taking the 8-byte aligned

address pointing to the doubleword beyond the 64-bit general register save area, adjusting it by subtracting 8 times the last (highest) 64-bit nonvolatile register number to be saved or restored and adding 8 × 13 = 104. These two adjustments allow positive offsets, and adjust so that the last register saved is placed directly below the 32-bit general register save area. These two adjustments allow a single routine, with fixed offsets, to be used across all potential cases. The doubleword beyond the 64-bit general-purpose register save area could be the low word of the 32-bit general-purpose register save area, the CR save word, or a pad word, depending on the number of 32-bit registers saved and the presence or absence of a CR save word.

ATR-SPE

These rules are summarized in the following table.

Table 3-33. SPE Save And Restore Rules

Function Type	r11 Contents
save & restore 32-bit values (r*M* - r31)	address of backchain
save & restore 64-bit values (r*M* - r31)	address of backchain (or pad word below CR save word if CR is saved) - 144
save & restore 64-bit values (r*M* - r*N*, where *N* != 32)	address of low end of 32-bit save area/CR save word/padding, adjusted by subtracting (8 × *N*) and adding 104.

3.3.4.1. Details about the Functions

Each function described in this section is a family of 18 functions with identical behavior except for the number and kind of registers affected.

ATR-SPE

The function names use the notation [32/64] to designate the use of a 32 for the 32-bit general-purpose register functions and a 64 for the 64-bit general-purpose register functions. The suffix *_m*; designates the portion of the name that would be replaced by the first register to be saved. That is, to save registers 18 through 31, call `_save32gpr_18()`.

There are two families of register saving functions:

- The following simple register saving functions save the indicated registers and return

```
_savegpr_m()
```

ATR-CLASSIC-FLOAT

```
_savefpr_m()
```

ATR-SPE

```
_save32gpr_m()
_save64gpr_m() and _save64gpr_ctr_m()
```

- The following GOT register saving functions do not return directly:

  ```
  _savegpr_m_g()
  ```

  ATR-CLASSIC-FLOAT

  ```
  _savefpr_m_g()
  ```

  ATR-SPE

  ```
  _save32gpr_m_g()
  _save64gpr_m_g() and _save64gpr_ctr_m_g()
  ```

  Instead these functions branch to `_GLOBAL_OFFSET_TABLE_-4`, relying on a `blrl` instruction at that address to return to the caller of the save function with the address of a *Global Offset Table* in the link register.

There are three families of register restoring functions.

- The following simple register restoring functions restore the indicated registers and return:

  ```
  _restgpr_m()
  ```

  ATR-CLASSIC-FLOAT

  ```
  _restfpr_m()
  ```

ATR-SPE

```
_rest32gpr_m() and _rest32gpr_m_t()
_rest64gpr_m() and _rest64gpr_ctr_m()
```

- The following exit functions restore the indicated registers and, relying on the registers being restored to be adjacent to the backchain word, restore the link register from the LR save word, remove the stack frame, and return through the link register:

```
_restgpr_m_x()
```

ATR-CLASSIC-FLOAT

```
_restfpr_m_x()
```

ATR-SPE

```
_rest32gpr_m_x()
_rest64gpr_m_x()
```

- The following tail functions restore the registers, place the LR save word into r0, remove the stack frame, and return to their caller:

```
_restgpr_m_t()
```

ATR-CLASSIC-FLOAT

```
_restfpr_m_t()
```

ATR-SPE

```
_rest64gpr_m_t()
```

The caller can thus implement a tail call by moving r0 into the link register and branching to the tail function. The tail function then detects the call from the function above the one that made the tail call and, when done, returns directly to it.

ATR-SPE

Note: There are no functions `_rest64gpr_ctr_m_x()` or `_reset64gpr_ctr_m_t()`, because the backchain word is not directly above the location of the 64-bit save area in these cases. In this case, the 64-bit registers shall be restored first, followed by a call to `_rest32gpr_m_x()` or `_rest32gpr_m_t()`.

Note: If a CR save word is used, even if only 64-bit registers are saved, `_rest64gpr_m_x()` and `rest64gpr_m_t()` cannot be used, because the backchain word is not directly above the end of the 64-bit save area.

ATR-SPE

The following assembly code shows an example of an implementation.

```
_save32gpr_14:        stw r14,-72(r11)
_save32gpr_15:        stw r15,-68(r11)
                ...
_save32gpr_30:        stw r30,-8(r11)
_save32gpr_31:        stw r31,-4(r11)
                      blr


_save64gpr_14:        evstdd r14,0(r11)
_save64gpr_15:        evstdd r15,8(r11)
                ...
_save64gpr_30:        evstdd r30,128(r11)
_save64gpr_31:        evstdd r31,136(r11)
                      blr


_save64gpr_ctr_14:    evstdd r14,0(r11)
                      bdz _save64gpr_ctr_done
_save64gpr_ctr_15:    evstdd r15,8(r11)
                      bdz _save64gpr_ctr_done
                ...
_save64gpr_ctr_30:    evstdd r30,128(r11)
                      bdz _save64gpr_ctr_done
_save64gpr_ctr_31:    evstdd r31,144(r11)
_save64gpr_ctr_done: blr


_rest32gpr_14:        lwz r14,-72(r11)
_rest32gpr_15:        lwz r15,-68(r11)
                ...
_rest32gpr_30:        lwz r30,-8(r11)
_rest32gpr_31:        lwz r31,-4(r11)
                      blr
```

```
_rest64gpr_14:       evldd r14,0(r11)
_rest64gpr_15:       evldd r15,8(r11)
                ...
_rest64gpr_30:       evldd r30,128(r11)
_rest64gpr_31:       evldd r31,136(r11)
                     blr


_rest64gpr_ctr_14:   evldd r14,0(r11)
                     bdz _rest64gpr_ctr_done
_rest64gpr_ctr_15:   evldd r15,8(r11)
                     bdz _rest64gpr_ctr_done
                ...
_rest64gpr_ctr_30:   evldd r30,128(r11)
                     bdz _rest64gpr_ctr_done
_rest64gpr_ctr_31:   evldd r31,136(r11)
_rest64gpr_ctr_done: blr
```

The GOT forms of the save routines (with a suffix of *_g*) all replace the `blr` with `b _GLOBAL_OFFSET_TABLE_ - 4`.

The exit forms of the restore routines (with a suffix of *_x*) perform the following tasks in place of the `blr`:

ATR-CLASSIC-FLOAT

```
_rest[fg]pr_m_x      replaces the blr with    lwz r0,4(r11)
                                              mr r1,r11
                                              mtlr r0
                                              blr
```

ATR-SPE

```
_rest32gpr_m_x       replaces the blr with    lwz r0,4(r11)
                                              mr r1,r11
                                              mtlr r0
                                              blr
_rest64gpr_m_x       replaces the blr with    lwz r0,148(r11)
                                              addi r1,r11,144
                                              mtlr r0
                                              blr
```

The tail functions (with a suffix of *_t*) are similar to the exit functions, except they skip the `mtlr` instruction.

ATR-SPE

Note: The CTR-based 64-bit restore functions cannot perform the exit and tail optimizations as implemented here, because the address of the backchain word and the return address are not at a fixed offset from r11.

Note: For slightly higher performance in the restore function variants, the `lwz` of r0 and the restore of r31 could be reordered (but the label for `_rest[32/64]gpr_31*()` shall now point to the `lwz` of r0, not the load of r31).

ATR-SPE

The following assembly source code provides and an example restore function variant using `_rest32gpr_m_x()`.

```
 ...
_rest32gpr_30_x:     lwz    r30,-8(r11)
_rest32gpr_31_x:     lwz    r0,4(r11)
                     lwz    r31,-4(r11)
                     mtlr   r0
                     mr     r1,r11   # Change to addi r1,r11,144
                                     # for _rest64gpr* blr
```

ATR-SPE

The following figure shows sample prologue and epilogue code with full saves of all the nonvolatile general-purpose registers (r14 through r25 as 64-bit, r26 through r31 as 32-bit) and a stack frame size of less than 32 KB. The variable *len* refers to the size of the stack frame. The example assumes that the function does not alter the nonvolatile fields of the CR register and does no dynamic stack allocation.

Note: The following code assumes that the size of the executable or shared object in which the code appears is small enough that a relative branch can reach from any part of the text section to any part of the *Global Offset Table* or the *Procedure Linkage Table*. Because relative branches can reach ± 32 MB, this restriction is not considered serious. See *Chapter 5* for more information.

```
function:
       mflr    r0                    # Save return addr in caller’s frame
       stw     r0,4(r1)              # . . .
       li      r0,12                 # Set up CTR with number of 64-bit
                                     # registers to save.
       mr      r11,r1                # Set up r11 with backchain pointer
       mtctr   r0
       stwu    r1,-len(r1)           # Establish new frame
```

```
bl      _save32gpr_26          # Save 32-bits of some GPRs
addi    r11,r11,-120           # Adjust r11 down 24 bytes to bottom
                               # of 32-bit area, and down another 96
                               # bytes for the offset
mflr    r31                    # Place GOT ptr in r31
bl      _save64gpr_ctr_14_g    # Save 64-bit nonvolatile GPRs and
                               # fetch the GOT ptr
                               # Save CR here if necessary
                               # Body of function
li      r0,12                  # Set up CTR with number of regs to
                               #  restore
mtctr   r0
addi    r11,r1,len-120         # Compute offset from low end of
                               # 32-bit save area
bl      _rest64gpr_ctr_14      # Restore 64-bit GPRs
                               # Restore CR here if necessary
addi    r11,r1,len             # Compute backchain word address
b       _rest32gpr_26_x        # Restore 32-bit GPRs and return
```

3.3.5. Profiling

This section describes how profiling (counting the number of times that a function is called) can be performed on the Power Architecture. Profiling is not required for ABI compliance. If profiling is supported, this implementation is one of those possible.

The code in Figure 3-24 can be inserted at the beginning of any function, before the execution of the prologue code. The following is a high-level explanation of this code.

- The link register is saved in the LR save word of the caller stack frame.
- The register r0 contains the address of the count variable, which is initialized to 0.
- The function, `_mcount()`, gets called. This function increments the count variable. It also needs to restore the link register to its original value so that it can handle the case where the profiled function does not save the link register itself.

Figure 3-24. Profiling Example

```
.function_mc:
               .data
               .align   2
               .long    0
               .text
function:
               mflr     r0
               addis    r11,r0,.function_mc@ha
               stw      r0,4(r1)
               addi     r0,r11,.function_mc@l
```

```
                bl        _mcount
```

NOTE: In the figure, the assembler expression symbol@l represents the lower-order 16 bits of the value for symbol. The assembly expression symbol@ha represents the higher-order 16 bits of the value for symbol, adjusted so that the addition of symbol@l and the shifted value of symbol@ha added together create the correct value of symbol. The adjustment is needed because symbol@l is a signed value.

3.3.6. Data Objects

Data objects with static storage duration are detailed here; stack resident data objects are omitted because the virtual address of stack resident data objects are derived relative to the stack or frame pointers.

The only instructions that can access memory in the Power Architecture are load and store instructions. Programs typically access memory by placing the address of the memory location into a register and accessing the memory location indirectly through the registers because Power Architecture instructions cannot hold 32-bit addresses directly. The values of symbols or their absolute virtual address are placed directly into instructions for symbolic references in absolute code.

Absolute addresses are not permitted in position-independent instructions. The signed offset into the *Global Offset Table* of the symbol is held in position-independent instructions that reference symbols. Then the absolute address of the table entry for the particular symbol can be derived by adding the offset to the appropriate *Global Offset Table* address using a general-purpose register. Figure 3-25 shows an example of this method, r31 loaded in the sample prologue.

Examples of absolute and position-independent compilations are shown in the following figures. These examples show the C language statements together with the generated assembly language. The assumption for the following figures is that only executables can use absolute addressing while shared objects can use position-independent code addressing. The figures are intended to demonstrate the compilation of each C statement independent of its context, hence there can be redundant operations in the code.

Figure 3-25. Absolute Load and Store Example

```
C code                            Assembly code
extern int src;                   .extern src
extern int dst;                   .extern dst
extern int *ptr;                  .extern ptr
                                  .section     ".text"
dst = src;                        lis 9,src@ha
                                  lwz 0,src@l(9)
                                  lis 9,dst@ha
                                  stw 0,dst@l(9)
ptr = &dst;                       lis 11,ptr@ha
                                  lis 9,dst@ha
                                  la 0,dst@l(9)
                                  stw 0,ptr@l(11)
*ptr = src;                       lis 9,ptr@ha
                                  lwz 11,ptr@l(9)
                                  lis 9,src@ha
                                  lwz 0,src@l(9)
```

```
                                   stw 0,0(11)
```

Note: The offset in the *Global Offset Table* where the value of the symbol is stored is given by the assembly syntax `symbol@got`. This syntax represents the address of the variable named *symbol*. The offset for this assembly syntax cannot be any larger than 16 bits. In cases where the offset is greater than 16 bits, the assembly syntax that is used is:

- High adjusted part of the offset: `symbol@got@ha`
- High part of the offset: `symbol@got@h`
- Low part of the offset: `symbol@got@l`

Figure 3-26. Small Model Position-Independent Load and Store

```
C code                             Assembly code
extern int src;                    .extern src
extern int dst;                    .extern dst
extern int *ptr;                   .extern ptr
                                   .section     ".text"
                                   # GOT pointer in r31
dst = src;                         lwz 9,src@got(31)
                                   lwz 0,0(9)
                                   lwz 9,dst@got(31)
                                   stw 0,0(9)
ptr = &dst;                        lwz 9,ptr@got(31)
                                   lwz 0,dst@got(31)
                                   stw 0,0(9)
*ptr = src;                        lwz 9,ptr@got(31)
                                   lwz 11,0(9)
                                   lwz 9,src@got(31)
                                   lwz 0,0(9)
                                   stw 0,0(11)
```

Figure 3-27. Large Model Position-Independent Load and Store

```
C code                             Assembly code
extern int src;                    .extern src
extern int dst;                    .extern dst
int *ptr;                          .extern ptr
                                   .section     ".text"
                                   # Assumes GOT pointer in r31
dst = src;                         addis     r6,r31,src@got@ha
                                   lwz       r6,src@got@l(r6)
                                   addis     r7,r31,dst@got@ha
                                   lwz       r7,dst@got@l(r7)
                                   lwz       r0,0(r6)
                                   stw       r0,0(r7)
ptr = & dst;                       addis     r6,r31,dst@got@ha
                                   lwz       r0,dst@got@l(r6)
                                   addis     r7,r31,ptr@got@ha
                                   lwz       r7,ptr@got@l(r7)
```

```
                                        stw       r0,0(r7)
*ptr = src;                             addis     r6,r31,src@got@ha
                                        lwz       r6,src@got@l(r6)
                                        addis     r7,r31,ptr@got@ha
                                        lwz       r7,ptr@got@l(r7)
                                        lwz       r0,0(r6)
                                        lwz       r7,0(r7)
                                        stw       r0,0(r7)
```

Analogous to the symbol `_SDA_BASE_` described in the SVR4 ABI, the symbol `_SDA2_BASE_` shall have a value such that the address of any byte in the ELF sections .PPC.EMB.sdata2 and .PPC.EMB.sbss2 is within a signed 16-bit offset of `_SDA2_BASE_`'s value. See *Section 4.5* for details.

The following description of putting data in sections .sdata, .sbss, .sdata2, .sbss2, .PPC.EMB.sdata0, and .PPC.EMB.sbss0 makes a distinction between defined and external variables. In a source file, a variable that is not stored on the stack is either a defined variable whose definition is in the file (e.g., `int Var;` in C) or an external variable that is accessed by code in the file but is not defined in the file (e.g., `extern int ExVar;`).

A high-level language processor, such as a compiler, shall have a means (e.g., an option) of generating an ELF file that conforms to the following rules.

- Sections .sdata, .sbss, and .sdata2 shall contain at least the following:
  - Entries for those defined variables that are globally visible scalars of size 8 or fewer bytes and whose values will not be changed outside of the program (which excludes C variables that are volatile).
    - Every such defined variable whose initial value is explicitly nonzero and might be changed by the program shall have a .sdata entry that represents the variable.
    - Every such defined variable whose value is initially 0 and might be changed shall have a .sbss entry or a .sdata entry that represents the variable.
    - If the relocatable object generated is not intended to be part of a shared object, every such variable whose value cannot be changed by the program (such as a C variable that is const but not volatile) shall have a .sdata2 entry that represents the variable; otherwise, such constant variables shall have .sdata or .sbss entries, as appropriate.
  - Entries produced by link editor resolution of relocation types (see *Section 4.13*).
- The only external variables accessed by the generated code as .sdata, .sbss, .sdata2, .sbss2, .PPC.EMB.sdata0, or .PPC.EMB.sbss0 entries shall be as follows:
  - External variables that are scalars of 8 or fewer bytes, whose values might be changed by the program and whose values will not be changed outside of the program, shall be accessed as .sdata or .sbss entries. So the address of such a variable will be within a 16-bit signed offset of `_SDA_BASE_`, which in a shared object is the same value as `_GLOBAL_OFFSET_TABLE_`, and otherwise is loaded in r13 by a conforming application.
  - When the relocatable object is not to be part of a shared object, external variables that are scalars of 8 or fewer bytes, whose values cannot be changed by the program and whose values will not be changed outside of the program, shall be accessed as .sdata2 or .sbss2 entries. In a shared object,

those constant external variables shall be accessed as .sdata or .sbss entries. So the address of such a variable, when not in a shared object, will be within a 16-bit signed offset of `_SDA2_BASE_`, which is loaded into r2 by a conforming application.

For example, consider generating a relocatable object that will not be part of a shared object from the following C code fragment.

```
         int                  i_sdata            = 1;
         const int            i_sdata2           = 2;
         int                  i_sbss_or_sdata;
         short                s_sbss_or_sdata    = 0;
extern double                 d_sdata_or_sbss;
extern const double           d_sdata2;
extern double                 d_any_sdata_or_sbss[50];
extern const float            f_any_sdata_or_sbss[200];
extern union my_union         u_any_sdata_or_sbss;
extern const volatile float   cvf_any_sdata_or_sbss;
         int                  i_any_sdata[100]   = { 3 };
static   struct my_struct     s_any_sdata        = { 4, 6 };
         volatile const float vcf_any_sdata[5]   = { 5 };
         int                  i_any_sbss_or_sdata[100];
static   struct my_struct     s_any_sbss_or_sdata;
         volatile const float vcf_any_sbss_or_sdata[25];
```

If the code fragment defines all globally visible variables, a C compiler when conforming to the previously defined rules would place i_sdata in .sdata, i_sdata2 in .sdata2, and i_sbss_or_sdata and s_sbss_or_sdata in either .sbss or .sdata, while at the same time generating code that accesses external variable d_sdata_or_sbss using an offset relative to the value of `_SDA_BASE_` (which is in r13), accesses d_sdata2 using an offset relative to `_SDA2_BASE_` (which is in r2), and does not access any other external variables as .sdata, .sbss, .sdata2, .sbss2, .PPC.EMB.sdata0, or .PPC.EMB.sbss0 entries.

3.3.7. Function Calls

Direct function calls are made in programs with the Power Architecture **bl** instruction. A `bl` instruction can reach 32 MB backwards or forwards from the current position due to a self-relative branch displacement in the instruction. Therefore the size of the text segment in an executable or shared object is constrained when a `bl` instruction is used to make a function call. As depicted in the figure following, the `bl` instruction is generally used by a compiler to call a function. Two possibilities exist for the location of the function with respect to the caller:

- The called function is in the same executable or shared object as the caller. In this case the symbol is resolved by the link editor and the `bl` instructions branches directly to the called function as in Figure 3-28.

Figure 3-28. Direct Function Call

```
C code                              Assembly code
---------------------------------------------------------------
extern void function();
function();                         bl function
```

- The called function is not in the same executable or shared object as the caller. In this case the symbol cannot be directly resolved by the link editor. The link editor generates a branch to glue code. Subsequently the dynamic linker changes the glue code to branch to the function requested by the caller. See *Procedure Linkage Table* in *Section 5.2.5*.

For indirect function calls, the address of the function to be called is placed in the CTR register and a `bctrl` instruction is used to perform the indirect branch as shown in *Figure 3-29*, *Figure 3-30*, and *Figure 3-31*.

Figure 3-29. Absolute Indirect Function Call

```
C Code                             Asm Code
---------------------------------------------------
extern void function();
extern void (*ptrfunc) ();
                            .section .text
ptrfunc = function;         lis   r11,ptrfunc@ha
                            lis   r9,function@ha
                            la    r0,function@l(r9)
                            stw   r0,ptrfunc@l(r11)
return (*ptrfunc)();        lis   r9,ptrfunc@ha
                            lwz   r0,ptrfunc@l(r9)
                            mtctr r0
                            bctrl
```

Branches less than or equal to ± 64 KB (16-bit signed offset ± 32 KB) may use small model addressing. Figure 3-30 demonstrates how to make an indirect function call using small model position-independent branching.

Figure 3-30. Small Model Position-Independent Indirect Function Call

```
C Code                      Asm Code
-------------------------------------------------------------------------
extern void function();
extern void (*ptrfunc) ();
                            .section .text
                            /* GOT pointer is in r11  */
ptrfunc = function;         lwz   r9,ptrfunc@got(r11)
                            lwz   r0,function@got(r11)
                            stw   r0,0(r9)

return (*ptrfunc)();        lwz   r9,ptrfunc@got(r11)
                            lwz   r0,0(r9)
                            mtctr r0
                            bctrl
```

Branches in excess of ± 64 KB must use large model addressing. Figure 3-31 demonstrates how to make an indirect function call using large model position-independent branching.

Figure 3-31. Large Model Position-Independent Indirect Function Call

```
C code                          Assembly code
---------------------------------------------------------------------
extern void function();
extern void (*ptrfunc) ();
                            .section .got
                            /* got_base is the start of the .got section */
                            /* offset -0x8000 from the GOT pointer.  */
                            .got_base = .+32768
                            .ptrfunc .long ptrfunc
                            .function .long function
                            .section        ".text"

                             /* GOT pointer in r10  */
ptrfunc=function             lwz 9,.ptrfunc@got-.got_base(r11)
                             lwz 0,.function@got-.got_base(r11)
                             stw 0,0(9)

(*ptrfunc) ()                lwz 9,.ptrfunc@got-.got_base(r11)
                             lwz 0,0(9)
                             mtctr 0
                             bctrl
```

3.3.8. Branching

The flow of execution in a program is controlled by the use of branch instructions. Branch instructions can jump to locations up to 32 MB in either direction since they hold a value with a 64 MB range that is relative to the current location of the program execution, which is defined by the architecture.

The following figure shows the model for branch instructions.

```
C code                         Assembly code
label:                         .L01:
   ...                              ...
   goto label;                      b .L01
```

Branch selection is provided in C with switch statements. An address table is used by the compiler to implement the switch statement selections in cases where the case labels satisfy grouping constraints. Details that are not relevant are not shown by the use of simplifying constraints in the examples that follow.

- r12 holds the selection expression.
- Case label constants begin at zero.

- The assembler names .Lcasei, .Ldefault, and .Ltab are used for the case labels, the default, and the address table respectively.

Absolute Switch Code

```
C code                        Assembly code
switch(j)                          cmplwi    r12, 4
{                                  bge       .Ldefault
   case 0:                         slwi      r12, 2
   ...                             addis     r12, r12, .Ltab@ha
   case 1:                         lwz       r0, .Ltab@l(r12)
   ...                             mtctr     r0
   case 3:                         bctr
   ...                             .rodata
   default:                   .Ltab:
   ...                             .long     .Lcase0
}                                  .long     .Lcase1
                                   .long     .Ldefault
                                   .long     .Lcase3
                                   .text
```

Position-Independent Switch Code, All Models

```
C code                         Assembly code
switch(j)                            cmplwi  r12, 4
{                                    bge     .Ldefault
   case 0:                           bl      .L1
   ...                         .L1: slwi     r12, 2
   case 1:                           mflr    r11
   ...                               addi    r12, r12,.Ltab-.L1
   case 3:                           add     r0, r12, r11
   ...                               mtctr   r0
   default:                          bctr
   ...                         .Ltab:
}                                    b       .Lcase0
                                     b       .Lcase1
                                     b       .Ldefault
                                     b       .Lcase3
```

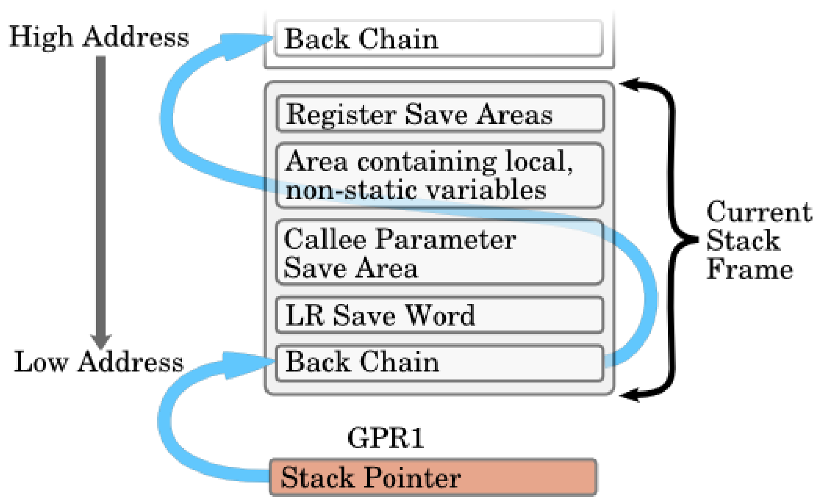
3.3.9. Dynamic Stack Space Allocation

When allocated, a stack frame may be grown or shrunk dynamically as many times as necessary across the lifetime of a function. Standard calling conventions must be maintained because a subfunction can be called after the current frame is grown and that subfunction may stack, grow, shrink, and tear down a frame between dynamic stack frame allocations of the caller. The following constraints apply when dynamically growing or shrinking a stack frame:

- Maintain 16-byte alignment.
- Stack pointer adjustments shall be performed atomically so that at all times the value of the backchain word is valid.
- Maintain addressability to the previously allocated local variables.

Note: Using a frame pointer is the recognized method for maintaining addressability to arguments or local variables. For correct behavior in the cases of `setjmp()` and `longjmp()` the frame pointer shall be allocated in a nonvolatile general-purpose register.

Figure 3-32. Before Dynamic Stack Allocation



An example organization of a stack frame before a dynamic allocation.

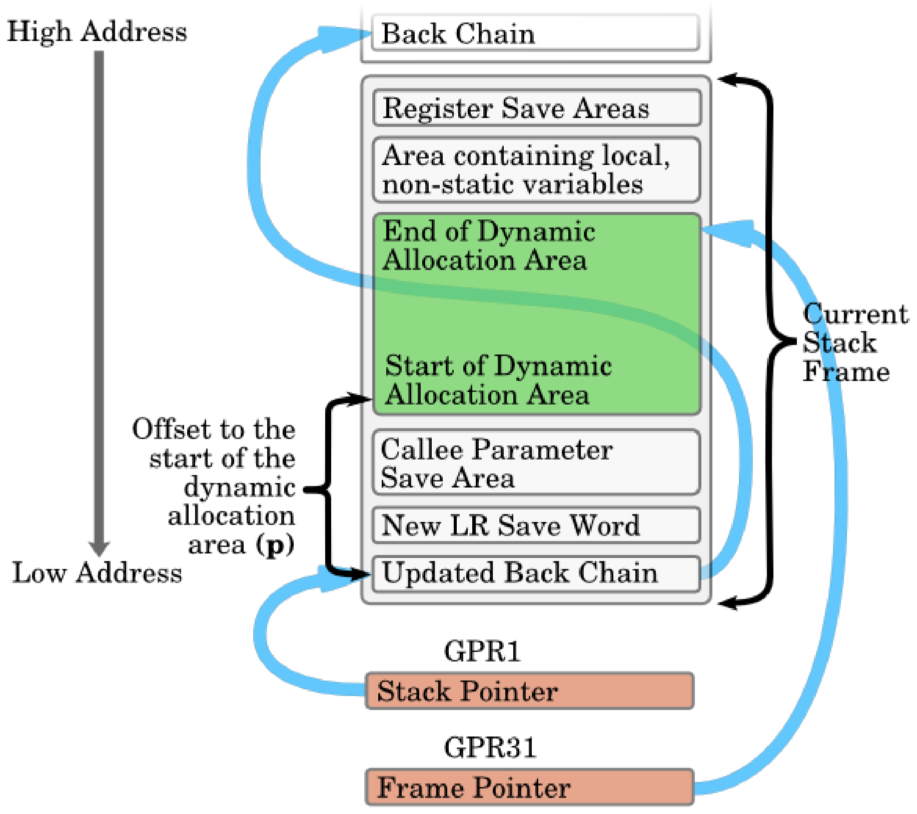
Figure 3-33. Example code to allocate n bytes:

```
#define n 13
char *a = alloca(n);
rnd(x) = round x to be multiple of stack alignment
psave = size of parameter save area (may be zero).
p = rnd(sizeof(psave+8)) ; Offset to the start of the dynamic allocation

lwz  0,0(1)            ; Load backchain word.
mr   31,1              ; Frame pointer to access previously allocated.
stwu 0,-rnd(n+15)(1)   ; Store new backchain, quadword-aligned.
addi 3,1,p             ; R3 = new data area following parmameter save area.
```

Note: Additional instructions might be needed to align the allocated data area or the stack pointer. Additional instructions will be necessary for an allocation of variable size.

Figure 3-34. After Dynamic Stack Allocation


High Address
Low Address
Back Chain
Register Save Areas
Area containing local, non-static variables
End of Dynamic Allocation Area
Start of Dynamic Allocation Area
Callee Parameter Save Area
New LR Save Word
Updated Back Chain
Current Stack Frame
Offset to the start of the dynamic allocation area (p)
GPR1
Stack Pointer
GPR31
Frame Pointer

An example organization of a stack frame after a dynamic allocation.

3.4. DWARF Definition

Although this ABI itself does not define a debugging format, DWARF (*Debug with Arbitrary Record Format*) (see *Section 1.1*) is defined here for systems that implement the DWARF specification.

The DWARF specification is used by compilers and debuggers to aid source-level or symbolic debugging. However, the format is not biased toward any particular compiler or debugger.

Per the DWARF specification, a mapping from Power Architecture registers to register numbers is required as described in Table 3-34.

Special Purpose Registers or SPRs are mapped into DWARF as 100 plus their SPR number. Performance Monitor Registers or PMRs are mapped into DWARF as 2048 plus the PMR number. Kernel debuggers that display privileged registers are to use the following DWARF register number mapping.

All instances of the Power Architecture use the following mapping for encoding registers into DWARF.

Table 3-34. Register Mappings

Register Name	Number	Abbreviation
General-purpose registers	0-31	R0-R31
Floating-point registers	32-63	F0-F31
Condition register	64	CR
Floating-point status and control register	65	FPSCR
Machine state register	66	MSR
Accumulator	99	ACC
SPRs	100-1123	LR, CTR, etc.
Vector registers	1124-1155	V0-V31
Reserved	1156-1199	
SPE high parts of GPRs	1200-1231	
Reserved	1232-2047	
Device control registers	3072-4095	DCRs
Performance monitor registers	4096-5120	PMRs

Chapter 4. Object Files

ATR-VLE

4.1. EABI Executable and Linking Format (ELF) Object Files

Implementations supporting VLE mark a per-page TLB entry storage control bit to indicate that a memory page holds either VLE Category **or** Embedded Category instructions. In this way the instructions in both the VLE category (*Book VLE*) and the Embedded Category (*Book III-E*) of the Power ISA can coexist in the same ELF binary.

Binding of VLE Category and Embedded Category memory pages to different memory bounds requires separation of VLE Category and Embedded Category encodings into different ELF sections, allowing easy identification for defining memory management page tables for run-time environments. Memory pages of VLE Category and Embedded Category instructions can be freely intermixed.

The VLE encodings also require additional relocation types (see relocations 216 - 233 in *Table 4-9*), which allow the link editor to resolve immediate and branch displacement fields in the instruction encoding once a symbol or label address is known (at link time).

ATR-VLE

4.2. EABI Object File Processing

An EABI-conforming link editor shall accept as input EABI-conforming and SVR4-conforming relocatable files, and it shall produce EABI-conforming shared object files.

4.3. ELF Header

The *file class* member of the ELF header identification array, `e_ident[EI_CLASS]`, identifies the ELF file as 32-bit encoded by holding the value 1, defined as class `ELFCLASS32`.

For a big-endian encoded ELF file the *data encoding* member of the ELF header identification array, `e_ident[EI_DATA]`, holds the value `2`, defined as data encoding `ELFDATA2MSB`. For a little-endian encoded ELF file it holds the value 1, defined as data encoding `ELFDATA2LSB`.

The ELF header `e_flags` member may hold the following bit masks that are applicable on the Power Architecture.

Table 4-1. e_flags Bit Masks

Mask	Value	Description
EF_PPC_EMB	0x80000000	Power Architecture Embedded Flag.
EF_PPC_RELOCATABLE_LIB	0x00008000	Mark ELF file as relocatable (containing Position Independent Code, see *Section 5.1.1*) and intended for use in a library.
EF_PPC_RELOCATABLE	0x00010000	Mark ELF file as relocatable (containing Position Independent Code, see *Section 5.1.1*).

EABI-conforming ELF files shall have `EF_PPC_EMB` set in the `e_flags` member.

The ELF header `e_machine` member identifies the architecture of the ELF file as the Power Architecture by holding the value 20, defined as machine name `EM_PPC`.

4.4. Special Sections

For the Power Architecture the following special sections with their corresponding section types and attributes apply:

.got

This section holds the *Global Offset Table* (GOT). Further information on accessing data in the GOT is contained in *Section 3.3.6*. Information on the layout of the Global Offset Table is in *Section 5.2.3*.

Name	Value
sh_name	.got
sh_type	SHT_PROGBITS
sh_flags	SHF_ALLOC + SHF_WRITE

.plt

This section holds the *Procedure Linkage Table* (PLT) (see *Section 5.2.5*).

ATR-BSS-PLT

Name	Value
sh_name	.plt
sh_type	SHT_NOBITS
sh_flags	SHF_ALLOC + SHF_WRITE + SHF_EXECINSTR

.sdata

Initialized data can be held in this section, which is part of the *Small Data Area* (SDA). Further information is found in *Section 4.8.1*.

Name	Value
sh_name	.sdata
sh_type	SHT_PROGBITS
sh_flags	SHF_ALLOC + SHF_WRITE

.sbss

Uninitialized data (set to zero on program execution) can be held in this section, which is part of the SDA (Small Data Area). Further information is found in *Section 4.8.1*.

Name	Value
sh_name	.sbss
sh_type	SHT_NOBITS
sh_flags	SHF_ALLOC + SHF_WRITE

.PPC.EMB.apuinfo

If an APU is required this section will contain records describing which are required for a program to execute properly. See *Section 4.10* for further details.

Name	Value
sh_name	.PPC.EMB.apuinfo
sh_type	SHT_NOTES
sh_flags	0

4.5. Special Embedded Sections

In addition to the special sections described in *Section 4.4*, an EABI-conforming ELF file shall be allowed to contain the following special sections. The SVR4 ABI has reserved for this document any section names beginning with .PPC.EMB.

.PPC.EMB.sdata2

This section holds initialized read-only small data that contributes to the program memory image. The section can, however, be used to hold writable data.

If a link editor creates a .PPC.EMB.sdata2 section that combines a .PPC.EMB.sdata2 section whose sh_flags is SHF_ALLOC with a .PPC.EMB.sdata2 section whose `sh_flags` is SHF_ALLOC +

SHF_WRITE, then the resulting .PPC.EMB.sdata2 section's `sh_flags` value shall be SHF_ALLOC + SHF_WRITE. See *Section 4.8.2* for more details.

Name	Value
sh_name	.PPC.EMB.sdata2
sh_type	SHT_PROGBITS
sh_flags	SHF_ALLOC or
	or
	SHF_ALLOC + SHF_WRITE
sh_link	SHF_UNDEF
sh_addralign	Maximum alignment required by any data item in .PPC.EMB.sdata2
sh_info	0
sh_entsize	0

.PPC.EMB.sbss2

The special section .PPC.EMB.sbss2 is intended to hold writable small data that contribute to the program memory image and whose initial values are 0. See *Section 4.8.2* for more details.

Name	Value
sh_name	.PPC.EMB.sbss2
sh_type	SHT_NOBITS
sh_flags	SHF_ALLOC + SHF_WRITE
sh_link	SHF_UNDEF
sh_addralign	Maximum alignment required by any data item in .PPC.EMB.sbss2.
sh_info	0
sh_entsize	0

.PPC.EMB.sdata0

This section is intended to hold initialized small data that contribute to the program memory image and whose addresses are all within a 16-bit signed offset of address 0. See *Section 4.8.3* for more details.

Name	Value
sh_name	.PPC.EMB.sdata0
sh_type	SHT_PROGBITS
sh_flags	SHF_ALLOC + SHF_WRITE
sh_link	SHF_UNDEF
sh_addralign	Maximum alignment required by any data item in .PPC.EMB.sdata0
sh_info	0
sh_entsize	0

.PPC.EMB.sbss0

This section is intended to hold small data that contribute to the program memory image, whose addresses are all within a 16-bit signed offset of address 0, and whose initial values are 0. See *Section 4.8.3* for further details.

Name	Value
sh_name	.PPC.EMB.sbss0
sh_type	SHT_NOBITS
sh_flags	SHF_ALLOC + SHF_WRITE
sh_link	SHF_UNDEF
sh_addralign	Maximum alignment required by any data item in .PPC.EMB.sbss0.
sh_info	0
sh_entsize	0

.PPC.EMB.seginfo

The special section .PPC.EMB.seginfo provides a means of naming and providing additional information about ELF segments (which are described by ELF program header table entries). A file shall contain at most one section named .PPC.EMB.seginfo. See *Section 4.12* for more details.

Name	Value
sh_name	.PPC.EMB.seginfo
sh_type	SHT_PROGBITS
sh_flags	0
sh_link	SHF_UNDEF or The section header table index of a section of type SHT_STRTAB whose string table contains the null terminated names to which entries in .PPC.EMB.seginfo refer.
sh_addr	0
sh_addralign	0
sh_info	0
sh_entsize	12

4.6. Symbol Table

4.6.1. Symbol Values

An executable file that contains a symbol reference that is to be resolved dynamically by an associated shared object will have a symbol table entry for that symbol. This entry will identify the symbol as undefined by setting the **st_shndx** member to **SHN_UNDEF**.

An executable file that needs to compare the value of two symbol references will have a symbol table entry for that symbol where the **st_value** member is nonzero.

If the **st_value** of an undefined symbol is nonzero, the loader must resolve every reference to the named symbol to the same value. This insures that all pointers to the symbol will be identical. If **st_value** is zero, the loader may resolve these symbols to different values, for example, to point directly to the symbol in some cases or into the GOT in other cases. If no PLT entry is allocated for the symbol, then **st_value** is zero.

ATR-BSS-PLT

Under the BSS-PLT ABI this **st_value** member holds the R_PPC_REL32 relocated address into the **.plt** section for the PLT entry used to resolve the undefined symbol. This PLT entry contains executable code used to dynamically resolve the address of the target symbol. The number of instructions in this code stub varies on the distance to the target.

Referencing GOT nonlocal statics is shown in Figure 3-26 and Figure 3-27. Taking the address of nonstatic function pointers is indicated by `<symbol>@plt`. Figure 3-30 and Figure 3-31 demonstrate how to perform this action.

4.8. EABI Small Data Areas

Three distinct small data areas, each possibly containing both initialized and zero-initialized data, are supported by the Embedded ABI, and are summarized in the following table.

Table 4-2. EABI Small Data Areas Summary

Section Names	Register or Value	Symbol	Shared Object Addressability?
.sdata .sbss	r13	`_SDA_BASE_`	local data only
.PPC.EMB.sdata2 .PPC.EMB.sbss2	r2	`_SDA2_BASE_`	no
.PPC.EMB.sdata0 .PPC.EMB.sbss0	0	n/a	no

In both shared objects and executables, the small data areas straddle the boundary between initialized and uninitialized data in the Data segment. The usual order of sections in the Data segment, some of which may be empty, is shown in Figure 4-3.

Figure 4-3. Section Ordering In the EABI

```
.rodata
.PPC.EMB.sdata2
.PPC.EMB.sbss2
.data
.got
.sdata
.sbss
.plt
.bss
```

All three small data areas can contain at most 64 KB of data items. All areas may hold both local and global data items in executables. In shared objects, .sdata/.sbss may only hold local data items, and the other two areas are not permitted. These areas are not permitted to hold values that might be changed outside of the program (that is, volatile variables).

Compilers may generate short-form, one-instruction references with 16-bit offsets for all data items that are in these six sections. Placing more data items in small data areas usually results in smaller and faster program execution.

These areas together provide up to 192 KB of data items that can be addressed in a single instruction: two 64-KB regions that can be placed anywhere in the address space but typically in standard locations (see *Section 4.8.1*), and one 64 KB region straddling address 0 (32 KB at addresses 0xFFFF_8000 through 0xFFFF_FFFF, and 32 KB at addresses 0x0000_0000 through 0x0000_7FFF).

Because the sizes of these areas are limited, compilers that support small data area relative addressing typically determine whether or not an eligible data item is placed in the small data area based on its size. Under this scheme, all data items less than or equal to a specified size (the default is usually 8 bytes) are placed in the small data area. Initialized data items are placed in one of the .data sections, uninitialized data items in one of the .sbss sections. If the default size results in a small data area that is too large to be addressed with 16-bit relative offsets, the link editor fails to build the executable file or shared object file, and some of the code that makes up the file must be recompiled with a smaller value for the size criterion.

This ABI does not preclude a compiler from using profiling information or some form of heuristics, rather than purely data item size, to make more informed decisions about which data items should be placed in these regions.

4.8.1. Small Data Area (.sdata and .sbss)

The small data area is part of the data segment of an executable program. It contains data items within the .sdata and .sbss sections, which can be addressed with 16-bit signed offsets from the base of the small data area.

Only data items with local (nonglobal) scope may appear in the small data area of a shared object. In a shared object the small data area follows the *Global Offset Table*, so data in the small data area can be addressed relative to the GOT pointer. However, in this case, the small data area is limited in size to no more than 32 KB, and less if the global offset table is large.

For executable files, up to 64 KB of data items with local or global scope can be placed into the small data area. In an executable file, the symbol `_SDA_BASE_` (small data area base) is defined by the link editor to be an address relative to which all data in the .sdata and .sbss sections can be addressed with 16-bit signed offsets or, if there is neither a .sdata nor a .sbss section, the value 0. In a shared object, `_SDA_BASE_` is defined to have the same value as `_GLOBAL_OFFSET_TABLE_`. The value of `_SDA_BASE_` in an executable is normally loaded into r13 at process initialization time, and r13 thereafter remains unchanged. In particular, shared objects shall not change the value in r13.

In executables, references to data items in the .sdata or .sbss sections are relative to r13; in shared objects, they are relative to a register that contains the address of the *Global Offset Table*.

4.8.2. Small Data Area 2 (.PPC.EMB.sdata2 and .PPC.EMB.sbss2)

Analogous to the symbol `_SDA_BASE_` described in the SVR4 ABI, the symbol `_SDA2_BASE_` shall have a value such that the address of any byte in the ELF sections .PPC.EMB.sdata2 and .PPC.EMB.sbss2 is within a signed 16-bit offset of `_SDA2_BASE_`'s value (see *Section 4.4*).

The sum of the sizes of sections .PPC.EMB.sdata2 and .PPC.EMB.sbss2 in an ELF file shall not exceed 64 KB. A file shall contain at most one section named .PPC.EMB.sdata2 and at most one section named .PPC.EMB.sbss2. In an executable file, data items with local or global scope can be placed into .PPC.EMB.sdata2 or .PPC.EMB.sbss2. Sections .PPC.EMB.sdata2 and .PPC.EMB.sbss2 shall not appear in a shared object.

If an executable file contains a .PPC.EMB.sdata2 section or a .PPC.EMB.sbss2 section, then a link editor shall set the symbol `_SDA2_BASE_` to be an address such that the address of any byte in .PPC.EMB.sdata2 or .PPC.EMB.sbss2 is within a 16-bit signed offset of `_SDA2_BASE_`. If an executable file does not contain .PPC.EMB.sdata2 or .PPC.EMB.sbss2, then a link editor shall set `_SDA2_BASE_` to 0.

If a link editor creates a .PPC.EMB.sdata2 section that combines a .PPC.EMB.sdata2 section whose `sh_flags` is SHF_ALLOC with a .PPC.EMB.sdata2 section whose `sh_flags` is SHF_ALLOC + SHF_WRITE, then the resulting .PPC.EMB.sdata2 section's `sh_flags` value shall be SHF_ALLOC + SHF_WRITE.

4.8.3. Small Data Area 0 (.PPC.EMB.sdata0 and .PPC.EMB.sbss0)

No symbol is needed for a base pointer for these sections (.PPC.EMB.sdata0 and .PPC.EMB.sbss0), because all addressing can be relative to address 0 (an address register encoding of r0 means the value 0 in Power Architecture load and store instructions).

The sum of the sizes of sections .PPC.EMB.sdata0 and .PPC.EMB.sbss0 in an ELF file shall not exceed 64 KB. A file shall contain at most one section named .PPC.EMB.sdata0 and at most one section named .PPC.EMB.sbss0. Data items with local or global scope can be placed into .PPC.EMB.sdata0 or .PPC.EMB.sbss0. Sections .PPC.EMB.sdata0 and .PPC.EMB.sbss0 shall not appear in a shared object.

4.9. DWARF Additions

In order to provide debuggers with the ability to identify where __ev64_opaque__ variables are located, several new DWARF operations have been added, as shown in the following table.

Table 4-3. DWARF Additions For `__ev64_opaque__` Support

Operation	Value	Description
DW_OP_ev64_opaque_reg***n***	0xe0-0xff	The data object addressed is in the upper and lower halves of register ***n***, where ***n*** is 0 through 31.

ATR-SPE

4.10. APU Information Section

This section allows disassemblers and debuggers to properly interpret the instructions within the binary, and could also be used by operating systems to provide emulation or error checking of the APU revisions. The format matches that of typical ELF note sections, as shown in *Table 4-4*.

Table 4-4. Typical Elf Note Section Format

length of name (in bytes)
length of data (in bytes)
type
name (null-terminated, padded to 4-byte alignment)
data

For the .PPC.EMB.apuinfo section, the name shall be *APUinfo\0*, the type shall be 2, and the data shall contain a series of words containing APU information, one per word as in *Table 4-5* and *Table 4-6*. The APU information contains two unsigned halfwords: the upper half contains the unique APU identifier, and the lower half contains the revision of that APU.

Table 4-5. Object File a.o

Offset	Value	Comment
0	0x00000008	8 bytes in "APUinfo\0"
4	0x0000000C	12 bytes (3 words) of APU information
8	0x00000002	NOTE type 2
12	"APUinfo\0"	string identifying this as APU information
20	0x00010001	APU #1, revision 1
24	0x00020003	APU #2, revision 3
28	0x00040001	APU #4, revision 1

Table 4-6. Object File b.o

Offset	Value	Comment
0	0x00000008	8 bytes in "APUinfo\0"
4	0x00000008	8 bytes (2 words) of APU information
8	0x00000002	NOTE type 2
12	"APUinfo\0"	string identifying this as APU information
20	0x00010002	APU #1, revision 2
24	0x00040001	APU #4, revision 1

Linkers shall merge all .PPC.EMB.apuinfo sections in the individual relocatable files into one, with merging of per-APU information as demonstrated in *Table 4-7*.

Table 4-7. Merged Object File b.o

Offset	Value	Comment
0	0x00000008	8 bytes in "APUinfo\0"
4	0x0000000C	12 bytes (3 words) of APU information
8	0x00000002	NOTE type 2
12	"APUinfo\0"	string identifying this as APU information
20	0x00010002	APU #1, revision 2
24	0x00020003	APU #2, revision 3
28	0x00040001	APU #4, revision 1

Note: It is assumed that a later revision of any APU is compatible with an earlier one, but the converse is not true. Thus, the resultant .PPC.EMB.apuinfo section requires APU #1 revision 2 or greater to work, and will not work on APU #1 revision 1. If an APU revision breaks backwards compatibility, it must obtain a new unique APU identifier.

Table 4-8. APU Identifiers

APU Identifier (16 Bits)	APU/Extension
0x003f	AltiVec
0x0040	ISEL
0x0041	PMR (Performance Monitor)
0x0042	RFMCI (Machine-check)
0x0043	CACHE_LOCK (Cache-locking)
0x0100	e500 SPE
0x0101	e500 SPFP/EFS
0x0102	e500 BRLOCK/BR_LOCK (Branch-locking/BTB locking)
0x0104	VLE
0x0000..0x003E	Reserved for legacy use
0x0044..0x00FF	Reserved

A link editor may optionally warn when different relocatable objects require different revisions of an APU, because moving the revision up may make the executable no longer work on processors with the older revision of the APU. In this example, the link editor could emit a warning like "Warning:bumping APU #1 revision number to 2, required by b.o."

ATR-VLE

4.11. VLE Identification

The executable and linking format (ELF) allows processor-specific section header and program header flag attributes to be defined. The following section header and program header flag attribute definitions are used to mark ELF sections containing VLE instruction encodings.

```
#define SHF_PPC_VLE 0x10000000            /* section header flag */
#define PF_PPC_VLE 0x10000000             /* program header flag */
```

The SHF_PPC_VLE flag marks ELF sections containing VLE instructions. Similarly, the PF_PPC_VLE flag is used by ELF program headers to mark program segments containing VLE instructions. If either the SHF_PPC_VLE flag or the PF_PPC_VLE flag is set, then instructions in those marked sections are interpreted as VLE instructions; Book E instructions reside in sections that do not have these flags set.

ELF sections setting the SHF_PPC_VLE flag that contain VLE instructions should also use the SHF_ALLOC and SHF_EXECINSTR bits as necessary. Setting the SHF_PPC_VLE bit does not automatically imply a section that is marked as allocate (SHF_ALLOC) or executable (SHF_EXECINSTR). The link editor keeps sections marked as VLE (SHF_PPC_VLE) in separate output sections that do not contain Book E instructions.

Similarly, ELF program headers setting the PF_PPC_VLE flag should use the PF_X, PF_W, and PF_R flags to indicate executable, writable, or readable attributes. It is considered an error for a program header with PF_PPC_VLE set to contain sections that do not have SHF_PPC_VLE set.

A program loader or debugger can then scan the section headers or program headers to detect VLE sections in case anything special is required for section processing or downloading.

ATR-VLE

4.12. ROM Copy Segment Information Section

Often embedded applications copy the initial values for variables from ROM to RAM at the start of execution. To facilitate this, a link editor resolves references to the application variables at their RAM locations, but relocates the variable's initial values to their ROM locations. An ELF segment whose raw data (addressed by the program header entry's p_offset field) consists of initial values to be copied to the locations of application variables is a ROM copy segment. One purpose of .PPC.EMB.seginfo is to define that one segment is a ROM copy of, and thus has the initial values for, a second segment.

The raw data for section .PPC.EMB.seginfo shall contain only 12-byte entries whose C structure is:

```
typedef struct {
    Elf32_Half sg_indx;
    Elf32_Half sg_flags;
    Elf32_Word sg_name;
    Elf32_Word sg_info;
} Elf32_PPC_EMB_seginfo;
```

where the structure members are defined as follows:

sg_indx

The index number of a segment in the program header table. Program header table entries are considered to be numbered from 0 to *n* - 1, where *n* is the number of table entries.

sg_flags

A bit mask of flags. The only allowed flag shall be as shown in the following table.

Flat Name	Value	Allowed Flag Meaning
PPC_EMB_SG_ROMCOPY	0x0001	Segment indexed by `sg_indx` is a ROM copy of the segment indexed by `sg_info`.

sg_name

The offset into the string table where the null terminated name for the segment indexed by `sg_indx`> is found. The section index of the string table to be used is in the `sh_link` field of .PPC.EMB.seginfo's section header. If `sh_link` is SHN_UNDEF, then sg_name shall be 0 for all .PPC.EMB.seginfo entries. An `sg_name` value of 0 shall mean that the segment indexed by `sg_indx` has no name.

sg_info

Contains information that depends on the value of `sg_flags`. If the flag PPC_EMB_SG_ROMCOPY is set in `sg_flags`, then `sg_info` shall be the index number of the segment for which the segment indexed by `sg_indx` is a ROM copy; otherwise, the value of `sg_info` shall be 0.

If one segment is a ROM copy of a second segment (based on information in section .PPC.EMB.seginfo), then:

- The first segment's p_type value shall be PT_LOAD.
- The second segment's p_type value shall be PT_NULL.
- Under EABI extended conformance none of the relocation entries that a dynamic linker might resolve shall refer to a location in the segment that is the ROM copy of another segment.

If the section exists, .PPC.EMB.seginfo shall contain at least one entry but need not contain an entry for every segment. Entries shall be in the same order as their corresponding segments in the ELF program header table (increasing values of sg_indx). Only one .PPC.EMB.seginfo entry shall be allowed per segment.

A link editor may support creation of section .PPC.EMB.seginfo, and, if it supports creation, it may support only segment naming, only ROM copy segments, or both.

4.13. Relocation Types

ATR-EABI-EXTENDED

Under the EABI support for dynamic linking, the GOT, and the PLT is considered EABI extended conformance.

The relocation entries in a relocatable file are used by the link editor to transform the contents of said file into an executable file or shared object file. The application and result of a relocation are similar for both. Several relocatable files may be combined into one output file. The link editor merges the content of the files, sets the value of all function symbols, and performs relocations.

The 32-bit Power Architecture uses `Elf32_Rela` relocation entries exclusively. A relocation entry may operate upon a halfword, word, or doubleword. The `r_offset` member of the relocation entry designates the first byte of the address affected by the relocation. The subfield of `r_offset` affected by a relocation is implicit in the definition of the applied relocation type. The `r_addend` member of the relocation entry serves as the relocation addend which is described per relocation formula.

A *relocation type* defines a set of instructions and calculations necessary to alter the subfield data of a particular relocation field.

4.13.1. Relocation Fields

The following *relocation fields* identify a subfield of an address affected by a relocation.

Bit numbers appear at the bottom of the boxes. Byte numbers appear in the top of the boxes; big-endian in the upper left corners and little-endian in the upper right corners. The byte order specified in a relocatable file's ELF header applies to all the elements of a relocation entry, the relocation field definitions, and relocation type calculations.

word32

Specifies a 32-bit bit-field taking up 4 bytes maintaining 4-byte alignment unless otherwise indicated.

Under the EABI, this field shall have no-alignment restrictions.

0 3	1 2	2 1	3 0
	word32		
0			31

word30

Specifies a 30-bit bit-field taking up bits 0-29 of a word, maintaining 4-byte alignment unless otherwise indicated.

0 3	1 2	2 1	3	0
	word30			
0			29	30 31

low24

Specifies a 24-bit bit-field taking up bits 6-29 of a word, maintaining 4-byte alignment. The other bits remain unchanged. A branch instruction is an example of this field.

0	3	1 2	2 1	3	0
		low24			
0 5	6			29	30 31

low21

Specifies a 21-bit bit-field occupying the least significant bits of a word with 4-byte alignment.

Under the EABI, this field shall have no-alignment restrictions.

0 3	1	2	2 1	3 0
			low21	
0 10				31

low14

Specifies a 14-bit bit-field taking up bits 16-29 and possibly bit 10 (branch prediction bit) of a word, maintaining 4-byte alignment. The other bits remain unchanged. A conditional branch instruction is an example usage.

0 3 | 1		2	2 1 | 3	0
			low14	
0	10	15	16 29	30 31

half16

Specifies a 16-bit bit-field taking up two bytes, maintaining 2-byte alignment. The immediate field of an Add Immediate instruction is an example of this field.

Under the EABI, this field shall have no-alignment restrictions.

0 1 | 1 0
half16
0 15

ATR-SPE

4.13.2. SPE Specific Relocation Fields

mid5

Specifies a 5-bit bit-field occupying the most significant bits of the least-significant halfword of a word with 4-byte alignment. This relocation field is used primarily for the SPE APU load/store instructions.

0 3 | 1 2	2	1 | 3 0
	mid5	
0 15	16 20	21 31

mid10

Specifies a 10-bit bit-field occupying bits 11 through 20 of a word with 4-byte alignment. This relocation field is used primarily for the SPE APU load/store instructions.

0 3 | 1	2 | 2	1 | 3 0
	mid10	
0 10	11 20	21 31

ATR-SPE

ATR-VLE

4.13.3. VLE Specific Relocation Fields

split20

20-bit bit-field with the 4 MSBs occupying bits 17 to 20, the next 5 bits occupying bits 11 to 15, and the remaining 11 bits occupying bits 21 to 31.

In addition, bits 0 to 5 in the destination word are encoded with the binary value 011100, bit 16 is encoded with the binary value 0.

Note: This relocation field specifies the opcode for the VLE e_li instruction, allowing the link editor to force the encoding of the e_li instruction, potentially changing the user's specified instruction. This functionality supports small data area relocation types. (R_PPC_VLE_SDA21 and R_PPC_VLE_SDA21_LO).

0 5	6 10	11 15	16	17 20	21 31
011100	---	split20	0	split20	split20
		4:8		0:3	9:19

split16a

16-bit bit-field with the 5 MSBs occupying bits 11 to 15 (the rA field) and the remaining 11 bits occupying bits 21 to 31.

0 10	11 15	16 20	21 31
---	split16a	---	split16a
	0:4		5:15

split16d

16-bit bit-field with the 5 MSBs occupying bits 6 to 10 (the rD field) and the remaining 11 bits occupying bits 21 to 31.

0 5	6 10	11 20	21 31
---	split16d	---	split16d
	0:4		5:15

bdh24

24-bit bit-field occupying bits 7 to 30 used to resolve branch displacements to half-word boundaries.

0 6	7 30	31
---	bdh24	--

bdh15

15-bit bit-field occupying bits 16 to 30 used to resolve branch displacements to half-word boundaries.

0 15	16 30	31
---	bdh15	--

bdh8

8-bit bit-field occupying bits 8 to 15 of a half-word. This field is used by a 16-bit branch instruction.

0 7	8 15
---	bdh8

ATR-VLE

4.13.4. Relocation Notations

The following notations are used in the relocation table.

A

Represents the addend used to compute the value of the relocatable field.

B

Represents the base address at which a shared object file has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different. See Program Header in the System V ABI for more information about the base address.

G

Represents the offset into the *Global Offset Table*, relative to the `_GLOBAL_OFFSET_TABLE_` symbol, at which the address of the relocation entry's symbol will reside during execution. This implies the creation of a **.got** section. See *Section 3.3* and the *Section 5.2.3* for more information.

Reference in a calculation to the value G implicitly creates a GOT entry for the indicated symbol.

L

Represents the section offset or address of the procedure linkage table entry for the symbol. This implies the creation of a **.plt** section if one does not already exist. It also implies the creation of a PLT entry for resolving the symbol. For an unresolved symbol the PLT entry points to a PLT resolver stub. For a resolved symbol a *Procedure Linkage Table* entry holds the final effective address of a dynamically resolved symbol (see *Section 5.2.5*).

P

Represents the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).

R

Represents the offset of the symbol within the section in which the symbol is defined (its section-relative address).

S

Represents the value of the symbol whose index resides in the relocation entry.

Represents the value of the symbol whose index resides in the relocation entry's `r_info` field.

T

Represents the offset from `_SDA_BASE_` to the location in the .sdata section that the link editor placed the address of the symbol whose index is in `r_info`. See the description for R_PPC_EMB_SDAI16 in *Section 4.13.6*.

U

Represents the offset from `_SDA2_BASE_` to the location in the .PPC.EMB.sdata2 section that the link editor placed the address of the symbol whose index is in `r_info`. See the description for R_PPC_EMB_SDA2I16 in *Section 4.13.6*.

V

Represents the offset to the symbol whose index is in `r_info` from the start of that symbol's containing section.

W

Represents the address of the start of the section containing the symbol whose index is in `r_info`.

X

Represents the offset from the appropriate base (`_SDA_BASE_`, `_SDA2_BASE_`, or 0) to where the link editor placed the symbol whose index is in `r_info`. This notation is generalized for the T and U cases.

Y

Represents a 5-bit value for the base register for the section where the link editor placed the symbol whose index is in `r_info`. Acceptable values are: the value 13 for symbols in .sdata or.sbss, the value 2 for symbols in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, or the value 0 for symbols in .PPC.EMB.sdata0 or .PPC.EMB.sbss0.

$+$

Denotes 32-bit modulus addition.

-

Denotes 32-bit modulus subtraction.

`>>`

Denotes arithmetic right-shifting.

`||`

Denotes concatenation of bits or bit-fields.

#lo(value)

Denotes the least significant 16 bits of the indicated value, i.e.,

```
#lo(x) = (x & 0xffff).
```

#hi(value)

Denotes bits 16 through 31 of the indicated value, i.e.,

```
#hi(x) = ((x >> 16) & 0xffff).
```

#ha(value)

Denotes the high adjusted value: bits 16 through 31 of the indicated value, compensating for #lo() being treated as a signed number, i.e.,

```
#ha(x) = (((x >> 16) + ((x & 0x8000) ? 1 : 0)) & 0xffff)
```

`_SDA_BASE_`

A symbol defined by the link editor whose value in shared objects is the same as `_GLOBAL_OFFSET_TABLE_`, and in executable programs is an address within the small data area.

`_SDA2_BASE_`

A symbol defined by the link editor whose value in executable programs is an address within the small data 2 area. See *Section 4.8* for more details.

`_BRTAKEN`
`_BRNTAKEN_`

Specify whether the branch prediction bit (bit 10) should indicate that the branch will be taken or not taken, respectively. For an unconditional branch, the branch prediction bit must be 0.

The following rules apply to the relocation types defined in the relocation table described later:

- For relocation types in which the names contain 14 or 16, the upper 17 bits of the value computed before shifting must all be the same. For relocation types whose names contain 24, the upper 7 bits of the value computed before shifting must all be the same. For relocation types whose names contain 14 or 24, the low 2 bits of the value computed before shifting must all be zero.

ATR-VLE

- For relocation types associated with branch displacements, in which the name of the relocation type contains 8, the upper 24 bits of the computed value before shifting must all be the same (either all zeros or all ones — that is, sign-extended displacement). For relocation types in which the name contains 15, the upper 17 bits of the computed value before shifting must all be the same. For relocation types in which the name contains 24, the upper 7 bits of the computed value before shifting must all be the same. For relocation types whose names contain 8, 15, or 24, the low 1-bit of the computed value before shifting must be zero (half-word boundary).

ATR-EABI-EXTENDED

- EABI optional relocation types are marked with a percentage symbol (%) after their name. These are provided for dynamic linking and for compatibility with existing vendor-defined relocations.

- The relocation types whose Field column entry contains an asterisk (*) are subject to failure if the value computed does not fit in the allocated bits.

4.13.5. Relocation Types Table

Table 4-9. Relocation Table

Relocation Name	Value	Field	Expression
R_PPC_NONE	0	none	none
R_PPC_ADDR32	1	word32	S + A
R_PPC_ADDR24	2	low24*	(S + A) >> 2
R_PPC_ADDR16	3	half16*	S + A
R_PPC_ADDR16_LO	4	half16	#lo(S + A)
R_PPC_ADDR16_HI	5	half16	#hi(S + A)
R_PPC_ADDR16_HA	6	half16	#ha(S + A)
R_PPC_ADDR14	7	low14*	(S + A) >> 2
R_PPC_ADDR14_BRTAKEN	8	low14*	(S + A) >> 2
R_PPC_ADDR14_BRNTAKEN	9	low14*	(S + A) >> 2
R_PPC_REL24	10	low24*	(S + A - P) >> 2
R_PPC_REL14	11	low14*	(S + A - P) >> 2
R_PPC_REL14_BRTAKEN	12	low14*	(S + A - P) >> 2
R_PPC_REL14_BRNTAKEN	13	low14*	(S + A - P) >> 2
R_PPC_GOT16	14	half16*	G
R_PPC_GOT16_LO	15	half16	#lo(G)
R_PPC_GOT16_HI	16	half16	#hi(G)
R_PPC_GOT16_HA	17	half16	#ha(G)
ATR-EABI-EXTENDED			
R_PPC_PLTREL24%	18	low24*	(L + A - P) >> 2

R_PPC_COPY	19	none	(see *Section 4.13.6*)
R_PPC_GLOB_DAT	20	word32	S + A (see *Section 4.13.6*)
R_PPC_JMP_SLOT	21	none	(see *Section 4.13.6*)
R_PPC_RELATIVE	22	word32	B+A (see *Section 4.13.6*)

ATR-EABI-EXTENDED

R_PPC_LOCAL24PC%	23	low24*	(see *Section 4.13.6*)

R_PPC_UADDR32	24	word32*	S + A (see *Section 4.13.6*)
R_PPC_UADDR16	25	half16*	S + A (see *Section 4.13.6*)
R_PPC_REL32	26	word32*	S + A - P
R_PPC_PLT32	27	word32*	L
R_PPC_PLTREL32	28	word32*	L-P
R_PPC_PLT16_LO	29	half16	#lo(L)
R_PPC_PLT16_HI	30	half16	#hi(L)
R_PPC_PLT16_HA	31	half16	#ha(L)
R_PPC_SECTOFF	33	half16*	R + A
R_PPC_SECTOFF_LO	34	half16	#lo(R + A)
R_PPC_SECTOFF_HI	35	half16	#hi(R + A)
R_PPC_SECTOFF_HA	36	half16	#ha(R + A)
R_PPC_ADDR30	37	word30	(S + A - P) >> 2

Table 4-10. Relocation Table - Continued

Relocation Name	Value	Field	Expression
	38 ... 66		Assigned to the PowerPC 64-bit ABI.
	67 ... 100		Assigned to the TLS ABI.
R_PPC_EMB_NADDR32	101	word32	(A - S)
R_PPC_EMB_NADDR16	102	half16*	(A - S)
R_PPC_EMB_NADDR16_LO	103	half16	#lo(A - S)
R_PPC_EMB_NADDR16_HI	104	half16	#hi(A - S)
R_PPC_EMB_NADDR16_HA	105	half16	#ha(A - S)
R_PPC_EMB_SDAI16	106	half16*	T (see *Section 4.13.6*)
R_PPC_EMB_SDA2I16	107	half16*	U (see *Section 4.13.6*)
R_PPC_EMB_SDA2REL	108	half16*	S + A-`_SDA2_BASE_`
R_PPC_EMB_SDA21	109	low21	Y || (X + A) (see *Section 4.13.6*)
R_PPC_EMB_MRKREF	110	none	(see *Section 4.13.6*)
R_PPC_EMB_RELSEC16	111	half16*	V + A
R_PPC_EMB_RELST_LO	112	half16	#lo(W + A)
R_PPC_EMB_RELST_HI	113	half16	#hi(W + A)
R_PPC_EMB_RELST_HA	114	half16	#ha(W + A)
R_PPC_EMB_BIT_FLD	115	word32*	(see *Section 4.13.6*)
R_PPC_EMB_RELSDA	116	half16	X + A (see *Section 4.13.6*)
	117 ... 179		Reserved for future use.
ATR-EABI-EXTENDED			
R_PPC_DIAB_SDA21_LO%	180	low21	Y || #lo(X + A)
R_PPC_DIAB_SDA21_HI%	181	low21	Y || #hi(X + A)
R_PPC_DIAB_SDA21_HA%	182	low21	Y || #ha(X + A)
R_PPC_DIAB_RELSDA_LO%	183	half16	#lo(X + A)
R_PPC_DIAB_RELSDA_HI%	184	half16	#hi(X + A)
R_PPC_DIAB_RELSDA_HA%	185	half16	#ha(X + A)

	186 ... 200		Reserved for future embedded system use.
	ATR-SPE		
R_PPC_EMB_SPE_DOUBLE	201	mid5*	(#lo(S + A)) >> 3
R_PPC_EMB_SPE_WORD	202	mid5*	(#lo(S + A)) >> 2
R_PPC_EMB_SPE_HALF	203	mid5*	(#lo(S + A)) >> 1
R_PPC_EMB_SPE_DOUBLE_SDAREL	204	mid5*	(#lo(S + A-`_SDA_BASE_`)) >> 3
R_PPC_EMB_SPE_WORD_SDAREL	205	mid5*	(#lo(S + A-`_SDA_BASE_`)) >> 2
R_PPC_EMB_SPE_HALF_SDAREL	206	mid5*	(#lo(S + A-`_SDA_BASE_`)) >> 1
R_PPC_EMB_SPE_DOUBLE_SDA2REL	207	mid5*	(#lo(S + A-`_SDA2_BASE_`)) >> 3
R_PPC_EMB_SPE_WORD_SDA2REL	208	mid5*	(#lo(S + A-`_SDA2_BASE_`)) >> 2
R_PPC_EMB_SPE_HALF_SDA2REL	209	mid5*	(#lo(S + A-`_SDA2_BASE_`)) >> 1
R_PPC_EMB_SPE_DOUBLE_SDA0REL	210	mid5*	(#lo(S + A)) >> 3
R_PPC_EMB_SPE_WORD_SDA0REL	211	mid5*	(#lo(S + A)) >> 2
R_PPC_EMB_SPE_HALF_SDA0REL	212	mid5*	(#lo(S + A)) >> 1
R_PPC_EMB_SPE_DOUBLE_SDA	213	mid10*	Y || ((#lo(X + A)) >> 3)
R_PPC_EMB_SPE_WORD_SDA	214	mid10*	Y || ((#lo(X + A)) >> 2)
R_PPC_EMB_SPE_HALF_SDA	215	mid10*	Y || ((#lo(X + A)) >> 1)

	ATR-VLE		
R_PPC_VLE_REL8	216	bdh8	(S + A - P) >> 1
R_PPC_VLE_REL15	217	bdh15	(S + A - P) >> 1
R_PPC_VLE_REL24	218	bdh24	(S + A - P) >> 1
R_PPC_VLE_LO16A	219	split16a	#lo(S + A)
R_PPC_VLE_LO16D	220	split16d	#lo(S + A)
R_PPC_VLE_HI16A	221	split16a	#hi(S + A)
R_PPC_VLE_HI16D	222	split16d	#hi(S + A)
R_PPC_VLE_HA16A	223	split16a	#ha(S + A)
R_PPC_VLE_HA16D	224	split16d	#ha(S + A)
R_PPC_VLE_SDA21	225	low21 split20	Y || (X + A) (see *Section 4.13.6*)
R_PPC_VLE_SDA21_LO	226	low21 split20	Y || #lo(X + A) (see *Section 4.13.6*)
R_PPC_VLE_SDAREL_LO16A	227	split16a	#lo(X + A)
R_PPC_VLE_SDAREL_LO16D	228	split16d	#lo(X + A)
R_PPC_VLE_SDAREL_HI16A	229	split16a	#hi(X + A)
R_PPC_VLE_SDAREL_HI16D	230	split16d	#hi(X + A)
R_PPC_VLE_SDAREL_HA16A	231	split16a	#ha(X + A)
R_PPC_VLE_SDAREL_HA16D	232	split16d	#ha(X + A)
R_PPC_VLE_ADDR20	233	split20	S + A
	234 ... 248		Reserved for future use.
	!ATR-SECURE-PLT		
	249 ... 252		Assigned for use by the Secure-PLT ABI.
	253 ... 255		Reserved for future use.

4.13.6. Relocation Descriptions

The following list describes relocations which can require special handling or description.

R_PPC_GOT16*

These relocation types resemble the corresponding R_PPC_ADDR16* types, except that they refer to the address of the symbol's *Global Offset Table* entry and additionally instruct the link editor to build a *Global Offset Table*.

R_PPC_PLTREL24

This relocation indicates that reference to a symbol should be resolved through a call to the symbol's *Procedure Linkage Table* entry. Additionally it instructs the link editor to build a procedure linkage table for the executable or shared object if one is not created.

ATR-BSS-PLT

Under the BSS-PLT ABI this relocation type may be implemented as a direct branch and link into the executable PLT slot which holds the absolute address (after resolution) of the specified symbol. There is an implicit assumption that the *Procedure Linkage Table* for a shared object or executable will be within $\pm$ 32 MB of an instruction that branches to it.

R_PPC_COPY

The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current relocatable file and in a shared object file. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.

R_PPC_GLOB_DAT

This relocation type resembles R_PPC_ADDR, except that it sets a *Global Offset Table* entry to the address of the specified symbol. This special relocation type allows determination of the correspondence between symbols and *Global Offset Table* entries.

R_PPC_JMP_SLOT

The link editor creates this relocation type for dynamic linking. Its offset member gives the location of a *Procedure Linkage Table* entry. The dynamic linker modifies the *Procedure Linkage Table* entry to transfer control to the designated symbol's address (see *Section 5.2.5*).

R_PPC_RELATIVE

The link editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_PPC_LOCAL24PC

This relocation type resembles R_PPC_REL24, except that it uses the value of the symbol within the object, not an interposed value, for S in its calculation. The symbol referenced in this relocation normally is `_GLOBAL_OFFSET_TABLE_`, which additionally instructs the link editor to build the *Global Offset Table*.

R_PPC_UADDR*

These relocation types are the same as the corresponding R_PPC32_ADDR* types, except that the datum to be relocated is allowed to be unaligned.

R_PPC_EMB_SDAI16

This instructs the link editor to create a 4-byte, word-aligned entry in the .sdata section containing the address of the symbol whose index is in the relocation entry's `r_info` field. At most one such implicit .sdata entry shall be created per symbol per link, and only in an executable file or shared object file. In addition, the value used in the relocation calculation shall be the offset from `_SDA_BASE_` to the symbol's implicit entry. The relocation entry's `r_addend` field value shall be 0.

R_PPC_EMB_SDA2I16

This instructs the link editor to create a 4-byte, word-aligned entry in the .PPC.EMB.sdata2 section containing the address of the symbol whose index is in the relocation entry's `r_info` field. At most one such implicit .PPC.EMB.sdata2 entry shall be created per symbol per link, and only in an executable file. In addition, the value used in the relocation calculation shall be the offset from `_SDA2_BASE_` to the symbol's implicit entry. The relocation entry's `r_addend` field value shall be 0.

ATR-SPE

R_PPC_EMB_SDA21

ATR-SPE

The most significant 11 bits at the address pointed to by the relocation entry shall be left unchanged.

The most significant 3 bits at the address pointed to by the relocation entry shall be left unchanged.

If the symbol whose index is in `r_info` is contained in .sdata or .sbss, then the link editor shall place in the next most significant 5 bits the value 13 (for r13); if the symbol is in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, then the link editor shall place in those 5 bits the value 2 (for r2); if the symbol is in .PPC.EMB.sdata0 or .PPC.EMB.sbss0, then the link editor shall place in those 5 bits the value 0 (for r0); otherwise, the link shall fail. The least significant 16 bits of this field shall be set to the address of the symbol plus the relocation entry's `r_addend` value minus the appropriate base for the symbol's section: `_SDA_BASE_` for a symbol in .sdata or .sbss, `_SDA2_BASE_` for a symbol in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, or 0 for a symbol in .PPC.EMB.sdata0 or .PPC.EMB.sbss0.

> Note: The source register in the `ori`, `oris`, `xor`, and `xoris` instructions (bits 6-10) are encoded differently than the `addi`, `addis`, `ld`, and `st` instructions (bits 11-15). This relocation type is appropriate for `add` and `ld` instructions, but not for `or` and `xor` instructions.

ATR-SPE

R_PPC_EMB_MRKREF

The symbol whose index is in `r_info` shall be in a different section from the section associated with the relocation entry itself. The relocation entry's `r_offset` and `r_addend` fields shall be ignored. Unlike other relocation types, the link editor shall not apply a relocation action to a location because of this type. This relocation type is used to prevent a link editor that does section garbage collecting from deleting an important but otherwise unreferenced section.

ATR-SPE

R_PPC_EMB_BIT_FLD

The most significant 16 bits of the relocation entry's `r_addend` field shall be a value between 0 and 31, representing a big-endian bit position within the entry's 32-bit location (e.g., 6 means the sixth most significant bit). The least significant 16 bits of `r_addend` shall be a value between 1 and 32, representing a length in bits. The sum of the bit position plus the length shall not exceed 32. The link editor shall replace bits starting at the bit position for the specified length with the value of the symbol, treated as a signed entity.

ATR-SPE

R_PPC_EMB_RELSDA

The link editor shall set the 16-bits at the address pointed to by the relocation entry to the address of the symbol whose index is in `r_info` plus the value of `r_addend` minus the appropriate base for the section containing the symbol: `_SDA_BASE_` for a symbol in .sdata or .sbss, `_SDA2_BASE_` for a symbol in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, or 0 for a symbol in .PPC.EMB.sdata0 or .PPC.EMB.sbss0. If the symbol is not in one of those sections, the link shall fail.

ATR-VLE

R_PPC_VLE_SDA21

The link editor computes a 21-bit value with the 5 MSBs having the value 13 (for r13), 2 (for r2), or 0. If the symbol whose index is in `r_info` is contained in .sdata or .sbss, the link editor supplies a value of 13; if the symbol is in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, the link editor supplies a value of 2; if the symbol is in .PPC.EMB.sdata0 or .PPC.EMB.sbss0, the link editor supplies a value of 0; otherwise, the link fails. The 16 least significant bits of this 21-bit value are set to the address of the symbol plus the relocation entry `r_addend` value minus the appropriate base for the symbol section:

- `_SDA_BASE_` for a symbol in .sdata or .sbss.

- `_SDA2_BASE_` for a symbol in .PPC.EMB.sdata2 or .PPC.EMB.sbss2.
- 0 for a symbol in .PPC.EMB.sdata0 or .PPC.EMB.sbss0.

If the 5 MSBs of the computed 21-bit value are nonzero, the link editor uses the low21 relocation field, where the 11 MSBs remain unchanged and the computed 21-bit value occupies bits 1131. Otherwise, the 5 MSBs of the computed 21-bit value are zero, with the following results:

- The link editor uses the split20 relocation field, where only bits occupying 610 remain unchanged.
- The 5 MSBs of the 21-bit value are ignored.
- The next most significant bit is copied to bit 11 and to bits 17 to 20 as a sign-extension.
- The next 4 most significant bits are copied to bits 12 to 15.
- The 11 remaining bits are copied to bits 21 to 31.
- In the destination word, bits 05 are encoded with the binary value 011100, and bit 16 is encoded with the binary value 0.

Note: Use of the split20 relocation field forces the encoding of the VLE e_li instruction, which can change the user's specified instruction.

ATR-VLE

R_PPC_VLE_SDA21_LO

Like R_PPC_VLE_SDA21, except that the #lo() operator obtains the 16 LSBs of the 21-bit value. The #lo() operator is applied after the address of the symbol plus the relocation entry `r_addend` value is calculated, minus the appropriate base for the symbol's section: `_SDA_BASE_` for a symbol in .sdata or .sbss, `_SDA2_BASE_` for a symbol in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, or 0 for a symbol in .PPC.EMB.sdata0 or .PPC.EMB.sbss0. The R_PPC_VLE_SDA21 entry describes applying the calculated 21-bit value to the destination word that uses either the low21 relocation field or the split20 relocation field.

Note: If the opcode is changed, 27 bits are changed instead of 21.

ATR-VLE

Note: The relocations R_PPC_VLE_SDA21 and R_PPC_VLE_SDA21_LO are not for load and store instructions (such as, e_lwz and e_stw), which should use the EABI relocation R_PPC_EMB_SDA21. These relocations, as written here, only start with an e_add16i. A link editor might convert the instruction to an e_li. Although other relocations do not specify the instructions they apply to, it may be useful to know that these relocations can apply only to one instruction.

4.14. EABI Relocations and Linking

An EABI conforming link editor shall support all of the relocations types in *Table 4-9* except for those listed in *Table 4-11*.

The relocatable fields of EABI relocation types shall have no alignment restrictions as indicated in *Section 4.13.1*.

Table 4-11. Relocation Types For EABI Extended Conformance

R_PPC_GOT16
R_PPC_GOT16_LO
R_PPC_GOT16_HI
R_PPC_GOT16_HA
R_PPC_PLT24
R_PPC_COPY
R_PPC_GLOB_DAT
R_PPC_JMP_SLOT
R_PPC_LOCAL24PC
R_PPC_PLT32
R_PPC_PLTREL32
R_PPC_PLT16_LO
R_PPC_PLT16_HI
R_PPC_PLT16_HA

ATR-EABI-EXTENDED

Under EABI extended conformance a link editor shall support all of the relocation types in *Table 4-9*, including those listed in *Table 4-11*, and a dynamic linker shall support all relocation types appropriate to dynamic linking.

A link editor shall not accept a relocation entry whose relocation type is not defined in *Table 4-9*.

ATR-EABI-EXTENDED

Under EABI extended conformance, a dynamic linker shall not process a relocation entry whose relocation type is not defined in *Table 4-9*.

Chapter 5. Program Loading and Dynamic Linking

5.3. EABI Program Loading and Dynamic Linking

Unlike the SVR4 ABI, an EABI-conforming entity shall not have program loading or program interpreter requirements.

An EABI-compliant ELF file contains absolute load addresses and sizes for each of its segments. There is no requirement that the dynamic linker follow the Read (PF_X), Write (PF_W), or Execute (PF_X) segment flags in the program header when loading the executable file.

Chapter 6. Libraries

6.1. Library Requirements

This ABI doesn't specify any additional interfaces for general-purpose libraries. However, certain processor specific support routines are defined in order to ensure portability between ABI conforming implementations.

Such processor specific support definitions concern floating-point alignment, register save/restore routines, variable argument list layout and a limited set of data definitions.

6.1.1. C Library Conformance with Generic ABI

6.1.1.1. Malloc Routine Return Pointer Alignment

The `malloc()` routine must always return a pointer with the alignment of the largest supported data type from the following list:

!ATR-LONG-DOUBLE-IBM && !ATR-DFP

- At least 8-byte (doubleword) aligned, as the returned pointer may be used for storing data items that require 8-byte alignment.

6.1.1.2. Library Handling of Limited-access Bits in Registers

Requirements for the handling of limited-access bits in certain registers by standard library functions are defined in *Section 3.2.1.2*.

6.1.2. Save and Restore Routines

All of the save and restore routines described in *Section 3.3.4* are required. These routines use unusual calling conventions due to their special purpose.

6.1.2.1. Save and Restore Routine Suffixes

The following suffix extensions describe the function templates in *Section 6.1.2.2*.

_m (*save and restore function variable*)

The variable *_m* represents the first register to be saved. That is, to save registers 18 to 31 using 32-bit saves, one would call save32gpr_18.

ATR-BSS-PLT

_g (*save function qualifier*)

GOT save functions are represented by the _g qualifier. These functions return to the caller of the save function by branching to the blrl instruction held at _GLOBAL_OFFSET_TABLE_-4.

_x (*restore function qualifier*)

Exit restore functions are represented by the _x qualifier. These functions restore the specified registers and use the link-register value in the calling function's LR-save area to return to the caller's parent function after removing the caller's stack frame.

_t (*restore function qualifier*)

Tail restore functions are represented by the _t qualifier. Given the following function call sequence where *function3* is a tail-call:

```
function1()
{
        function2();
        <further calls and code>
        return;
}

function2()
{
        _rest*_t();
        return function3();
}
```

The **tail restore functions** are called from *function2* and prepare the register state in *function2* for a tail-call to *function3* that is to return directly to *function1*. They restore the specified registers for *function1* from *function1's* stack frame and save the address of *function1* from the LRSAVE word of *function1*'s stack frame into R0 before returning control to *function2*. *Function2* then sets the LR to the address of *function1* held in R0 and calls the tail function *function3*. *Function3* will perform it's duty and then return directly to *function1* rather than *function2*.

ATR_SPE

_ctr (*save & restore function qualifier*)

CTR register save and restore functions are represented by the *_ctr* qualifier. These functions set the number of registers to be "saved to" or "restored from" into the CTR register.

6.1.2.2. Save and Restore Routine Templates

- _savegpr_m

!ATR-SECURE-PLT

- _savegpr_m_g

ATR-CLASSIC-FLOAT

- _savefpr_m

ATR-CLASSIC-FLOAT && !ATR-SECURE-PLT

- _savefpr_m_g

ATR-CLASSIC-FLOAT

- _restfpr_m

ATR-CLASSIC-FLOAT

- _restfpr_m_x

ATR-CLASSIC-FLOAT

- _restfpr_m_t

- _restgpr_m
- _restgpr_m_x
- _restgpr_m_t

ATR-SPE

- _save32gpr_m

ATR-SPE

- _save64gpr_m

ATR-SPE

- _save64gpr_ctr_m

ATR-SPE && !ATR-SECURE-PLT

- _save32gpr_m_g

ATR-SPE && !ATR-SECURE-PLT

- _save64gpr_m_g

ATR-SPE && !ATR-SECURE-PLT

- _save64gpr_ctr_m_g

ATR-SPE

- _rest32gpr_m

ATR-SPE

- _rest64gpr_m

ATR-SPE

- _rest64gpr_ctr_m

ATR-SPE

- _rest32gpr_m_x

ATR-SPE

- _rest64gpr_m_x

ATR-SPE

- _rest32gpr_m_t

ATR-SPE

- _rest64gpr_m_t

6.1.3. Types Defined In Standard Header

The type va_list shall be defined as follows:

```
typedef struct __va_list_tag {
  unsigned char gpr;
  unsigned char fpr;
  /* Two bytes padding.  */
  char *overflow_arg_area;
  char *reg_save_area;
} va_list[1];
```

The names and types of the elements are not part of the ABI, but the `__va_list_tag` name is part of the ABI (since it affects C++ name mangling), and the structure must have the size, alignment and layout implied by this definition.

- The **gpr** element holds the index of the next general-purpose register saved in this area from which an argument would be retrieved with `va_arg()`, where *gpr* == N corresponds to r$N + 3$. (If the argument is passed as DUAL_GP and *gpr* is odd, the next argument would be retrieved from r$N + 4$ and r*N* & plus; 5 instead.) If *gpr* is greater than 7, no more arguments will be retrieved from general-purpose registers by `va_arg()`.

ATR-CLASSIC-FLOAT

- The **fpr** element holds the index of the next floating-point register saved in this area from which an argument would be retrieved with `va_arg()`.
  - Fpr == *N* corresponds to f$N + 1$. If fpr is greater than 7, no more arguments will be retrieved from floating-point registers by `va_arg()`.

- **reg_save_area** points to an 8-byte-aligned area where registers r3 to r10 are saved, in that order.

  Addresses in the area pointed to by **reg_save_area** that correspond to registers used for passing named arguments, or to unused registers between those used for passing named arguments, need not correspond to allocated memory; those registers need not be saved in this area. va_arg shall only access those words required to load the argument of the type passed.

  ATR-SPE

  Only the low 32 bits of each register are saved in this area.

  ATR-CLASSIC-FLOAT

  Registers f1 to f8 immediately follow registers r3 to r10, if CR bit 6 was set when the variable-argument function was called.

- The **overflow_arg_area** element points to the word on the stack at the start of the next argument passed on the stack, or to a prior word that forms part of the padding required for the next argument to have the required alignment. va_arg shall only access those words required to load the argument of the type passed; if no arguments were passed on the stack, this area may not be allocated.

The following integer types are defined in headers required to be provided by freestanding implementations, or have their limits defined in such headers, and shall have the following definitions.

> Note: Freestanding implementations need not provide the types sig_atomic_t and wint_t.

- typedef int ptrdiff_t;
- typedef unsigned int size_t;
- typedef long wchar_t;

- typedef int sig_atomic_t;
- typedef unsigned int wint_t;
- typedef signed char int8_t;
- typedef short int16_t;
- typedef long int32_t;
- typedef long long int64_t;
- typedef unsigned char uint8_t;
- typedef unsigned short uint16_t;
- typedef unsigned long uint32_t;
- typedef unsigned long long uint64_t;
- typedef signed char int_least8_t;
- typedef short int_least16_t;
- typedef long int_least32_t;
- typedef long long int_least64_t;
- typedef unsigned char uint_least8_t;
- typedef unsigned short uint_least16_t;
- typedef unsigned long uint_least32_t;
- typedef unsigned long long uint_least64_t;
- typedef int int_fast8_t;
- typedef int int_fast16_t;
- typedef int int_fast32_t;
- typedef long long int_fast64_t;
- typedef unsigned int uint_fast8_t;
- typedef unsigned int uint_fast16_t;
- typedef unsigned int uint_fast32_t;
- typedef unsigned long long uint_fast64_t;
- typedef int intptr_t;
- typedef unsigned int uintptr_t;
- typedef long long intmax_t;
- typedef unsigned long long uintmax_t;

Appendix A. Taxonomy

The following list describes the archetypal ABI attributes used to conditionally define elements of the ABI. The relationship of these attributes is described in the taxonomy diagram in Figure A-1. A combination of these attributes is used to generate the individual Linux and Embedded ABI documents. These combinations are described in Appendix B. Each attribute description indicates whether it is an ABI software feature or an attribute that is tied to a specific Power ISA category.

32-bit PowerPC Archetypal ABI Attributes

ATR-BSS-PLT

(ABI Software Feature)

The text under this attribute defines the *BSS Procedure Linkage Table* ABI, which has a writable and executable PLT. **ATR-BSS-PLT is mutually exclusive with ATR-SECURE-PLT.**

ATR-CLASSIC-FLOAT

(Power ISA Category: Floating-Point)

The text under this attribute describes the classic Power Architecture floating-point ABI where there are 64-bit floating-point registers and an instruction set that accompanies them. **ATR-CLASSIC-FLOAT is mutually exclusive with ATR-SOFT-FLOAT.**

ATR-PASS-COMPLEX-IN-GPRS

(ABI Software Feature)

The text under this attribute describes a method for passing complex data types in GPRS. **ATR-PASS-COMPLEX-IN-GPRS is mutually exclusive and incompatible with ATR-PASS-COMPLEX-AS-STRUCT.** ATR-PASS-COMPLEX-IN-GPRS is predicated on ATR-CLASSIC-FLOAT or ATR-SOFT-FLOAT.

ATR-PASS-COMPLEX-AS-STRUCT

(ABI Software Feature)

The text under this attribute describes a method for passing complex data types as structures. **ATR-PASS-COMPLEX-AS-STRUCT is mutually exclusive and incompatible with ATR-PASS-COMPLEX-IN-GPRS.** ATR-PASS-COMPLEX-IN-GPRS is predicated on ATR-CLASSIC-FLOAT or ATR-SOFT-FLOAT.

ATR-CXX

(ABI Software Feature)

The text under this attribute describes C++ exception support as it impacts this ABI.

ATR-DFP

(Power ISA Category: Decimal Floating-Point)

The text under this attribute describes the *Decimal Floating Point* ABI as it relates to decimal floating-point registers, alignment, parameter passing, etc. This was introduced in Power ISA 2.05. ATR-DFP is predicated on ATR-CLASSIC-FLOAT or ATR-SOFT-FLOAT.

ATR-EABI

(Power ISA Category: Embedded)

This attribute describes elements that apply to the Embedded ABI as a whole.

ATR-EABI-EXTENDED

(ABI Software Feature)

This attribute describes elements that apply an implementation of the Embedded ABI with extended conformance such as support for dynamic linking, the GOT, PLT, full relocation support, etc.

ATR-LINUX

(Power ISA Category: Server)

This attribute describes elements that apply to the Linux ABI as a whole.

ATR-LONG-DOUBLE-IBM

(ABI Software Feature)

The text under this attribute describes usage of the AIX 128-bit Long Double format. **ATR-LONG-DOUBLE-IBM is mutually exclusive with ATR-LONG-DOUBLE-IS-DOUBLE.** ATR-LONG-DOUBLE-IBM is predicated on ATR-CLASSIC-FLOAT or ATR-SOFT-FLOAT.

ATR-LONG-DOUBLE-IS-DOUBLE

(ABI Software Feature)

The text under this attribute describes long double ABI when long double is treated as double. **ATR-LONG-DOUBLE-IS-DOUBLE is mutually exclusive with ATR-LONG-DOUBLE-IBM.** ATR-LONG-DOUBLE-IS-DOUBLE is predicated on ATR-CLASSIC-FLOAT or ATR-SOFT-FLOAT.

ATR-SECURE-PLT

(ABI Software Feature)

The text under this attribute describes the *Secure Procedure Linkage Table* ABI, which has a readable and writable, but nonexecutable PLT. **ATR-SECURE-PLT is mutually exclusive with ATR-BSS-PLT.**

ATR-SOFT-FLOAT

(ABI Software Feature)

The text under this attribute describes a software emulated 64-bit (double) floating-point ABI which also describes the conventions for *Embedded Floating Point* in 64-bit GPRs such as SPE-Float. **ATR-SOFT-FLOAT is mutually exclusive with ATR-CLASSIC-FLOAT.**

ATR-SPE

(Power ISA Category: SPE)

The text under this attribute describes the *Signal Processing Engine* ABI for the SPE facility that was introduced in Power ISA v2.03 It is a SIMD instruction set using two element short vectors within 64-bit GPRs. **ATR-SPE is mutually exclusive with ATR-VECTOR. ATR-SPE includes SPE-Float which leverages ATR-SOFT-FLOAT.** Therefore ATR-SPE is predicated on ATR-SOFT-FLOAT and mutually exclusive with ATR-CLASSIC-FLOAT.

ATR-TLS

(ABI Software Feature)

The text under this attribute describes the *Thread Local Storage* ABI. **At the time of this writing ATR-TLS is mutually exclusive with ATR-EABI** since ATR-EABI uses the thread local storage register for the SDATA2 pointer.

ATR-VECTOR

(Power ISA Category: Vector)

The text under this attribute describes the AltiVec and VMX float and integer SIMD instruction set ABI. **ATR-VECTOR is mutually exclusive with ATR-SPE.** ATR-VECTOR is predicated on ATR-CLASSIC-FLOAT or ATR-SOFT-FLOAT.

ATR-VLE

(Power ISA Category: VLE)

The text under this attribute describes the *Variable Length Encoding* environment as introduced in Power ISA 2.03.

The following taxonomy (described in EBNF) describes the relationship between the aforementioned ABI attributes.

Figure A-1. Taxonomy

```
ABI -> CommonCore OperatingEnvironment ISA-Flavor

CommonCore -> SYS-V-Without-Float { /* No attribute. Implicit. */ }

OperatingEnvironment -> Linux { atr = ATR-LINUX }
                     |  EABI  { atr = ATR-EABI  }

ISA-Flavour -> SIMD Encoding Floating-Point

SIMD -> Vector        { atr = ATR-VECTOR }
     |  SPE           { atr = ATR-SPE }
     |  /* Epsilon */ { com = "/* No SIMD. */" }

Encoding -> VLE { atr = ATR-VLE }

Floating-Point -> Common-Float Long-Double FP-Decimal

Common-Float -> Classic-Float-Common { atr = ATR-CLASSIC-FLOAT }
             |  Soft-Float-Common    { atr = ATR-SOFT-FLOAT }

Procedure-Linkage-Table -> BSS-PLT    { atr = ATR-BSS-PLT }
                        |  Secure-PLT { atr = ATR-SECURE-PLT }

Thread-Local-Storage -> TLS { atr = ATR-TLS }

Long-Double -> IBM  { atr = ATR-LONG-DOUBLE-IBM  }
             | None { atr = ATR-LONG-DOUBLE-IS-DOUBLE }

FP-Decimal -> /* Epsilon */ { com = "/* No FP-Decimal */" }
```

```
           |  DFP           { atr = ATR-DFP }

Complex -> Pass Complex in GPRS   { atr = ATR-PASS-COMPLEX-IN-GPRS }
        |  Pass Complex As Struct { atr = ATR-PASS-COMPLEX-AS-STRUCT }

CXX -> C++ Exception Handling  { atr = ATR-CXX }

EABI-Extended -> /* Epsilon */            { com = "/* No EABI Extended */" }
              | EABI Extended Conformance { atr = ATR-EABI-EXTENDED }
```

Appendix B. Attribute Inclusion and ABI Conformance

This appendix describes ABI attribute inclusion and conformance rules. It uses the attribute tags described in Appendix A.

B.1. ATR-LINUX Inclusion and Conformance

Linux ABI Attribute Inclusions:

- ATR-BSS-PLT
- ATR-CLASSIC-FLOAT
- ATR-CXX
- ATR-DFP
- ATR-LONG-DOUBLE-IBM
- ATR-LONG-DOUBLE-IS-DOUBLE
- ATR-SECURE-PLT
- ATR-SOFT-FLOAT
- ATR-SPE
- ATR-TLS
- ATR-VECTOR
- ATR-PASS-COMPLEX-IN-GPRS

Linux ABI Attribute Exclusions:

- ATR-PASS-COMPLEX-AS-STRUCT
- ATR-VLE
- ATR-EABI-EXTENDED

Linux ABI Conformance

- An implementation of the Linux ABI **must** implement at least one of the following:
  ATR-SOFT-FLOAT   ATR-CLASSIC-FLOAT
- If an implementation supports 64-bit vector types on SPE processors or uses the high parts of registers on such processors it must implement ATR-SPE.
- An implementation of the Linux ABI **must** implement ATR-LONG-DOUBLE-IBM and may also implement ATR-LONG-DOUBLE-IS-DOUBLE. A conforming application only uses one or the other.
- An implementation that supports decimal floating point **must** implement ATR-DFP. Hardware support for DFP requires implementation of ATR-CLASSIC-FLOAT otherwise ATR-SOFT-FLOAT can provide software emulation.
- An implementation **must** implement ATR-SECURE-PLT. ATR-BSS-PLT should be supported for

binary compatibility with previous versions of this ABI.

- Availability of Vector data types is subject to conformance to a Power ISA category where the categories *Vector* and *Signal Processing Engine* are mutually exclusive. A conforming application only uses ATR-VECTOR or ATR-SPE.

Note: An implementation of this ABI shall indicate explicitly which attributes are supported. Supporting attributes which are mutually exclusive is fine as long as only one is supported at a given time during application execution.

B.2. ATR-EABI Inclusion and Conformance

EABI Attribute Inclusions

ATR-BSS-PLT

ATR-CLASSIC-FLOAT

ATR-EABI-EXTENDED

ATR-PASS-COMPLEX-AS-STRUCT

ATR-PASS-COMPLEX-IN-GPRS

ATR-LONG-DOUBLE-IS-DOUBLE

ATR-SOFT-FLOAT

ATR-SPE

ATR-VLE

EABI Attribute Exclusions

ATR-CXX

ATR-DFP

ATR-LONG-DOUBLE-IBM

ATR-SECURE-PLT

ATR-TLS

ATR-VECTOR

EABI Conformance

- The EABI does not support thread local storage (ATR-TLS) at this time.
- The EABI does not support ATR-SECURE-PLT at this time.
- The EABI does not support *unwind information*.
- An implementation of the EABI ABI can implement ATR-PASS-COMPLEX-AS_STRUCT and/or implement ATR-PASS-COMPLEX-IN-GPRS but a conforming application shall only use one or the other.

- Conformance with the EABI does not require implementation of ATR-EABI-EXTENDED, which describes implementation of extended conformance such as support for dynamic linking, the GOT, PLT, full relocation support, etc.

> Note: An implementation of this ABI shall indicate explicitly which attributes are supported. Supporting attributes which are mutually exclusive is fine as long as only one is supported at a given time during application execution.

Appendix C. APUs and Power ISA Categories

This appendix discusses the relationship between *Auxiliary Processing Units* (APUs) and Power ISA categories.

APUs are a method used to extend the Power Architecture beyond the facilities described and ratified in the Power ISA. Since the adoption of the Power ISA many technologies that were historically presented as APUs have now been subsumed into the Power ISA as optional categories or phased into the base ISA.

Since this ABI is not predicated on minimum Power ISA version it continues to present information on APUs (see *Section 4.10*) that have been subsumed into the Power ISA. It is up to the implementation whether to follow the Power ISA or the APU designation based upon compatibility requirements and to specify APU information as necessary.

The following table identifies APUs and their relationship to the Power ISA.

Table C-1. APU Extensions and Corresponding Power ISA Categories

APU Extension	APU Identifier	Power ISA Category	Description
Altivec	0x003f	V	Vector Facility
PMR	0x0041	E.pm E	Embedded.Performance Monitor
RFMCI	0x0042	E	Embedded, Return From Machine Check Interrupt instruction
CACHE_LOCK	0x0043	ECL	Embedded Cache Locking
SPE	0x0100	SP, SP.FV	Signal Processing Engine, SPE.Embedded Float Vector
E500 SFFP/EFS	0x0101	SP.fs, SP.fd	Embedded Float Scalar Single, Embedded Float Scalar Double
VLE	0x0104	VLE	Variable Length Encoding
ISEL	0x0040	Base	Power ISA Base (mandatory), Integer Select instruction

The following APUs remain unspecified by the Power ISA (as of version 2.05).

Table C-2. APUs

APU Extension	APU Identifier
e500 BRLOCK	0x0102