E500ABIUG/D
3/2003
Rev. 1.0

PowerPC™ e500 Application Binary Interface User's Guide

Home Page:
www.freescale.com
email:
support@freescale.com
USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
480-768-2130
support@freescale.com
Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com
Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 2666 8080
support.japan@freescale.com
Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com
For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
(800) 441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

freescale™
semiconductor

Contents

Contents

Chapter 3
Object Files

Contents

Contents

Figures

Figures

Tables

Tables

Chapter 1
Introduction

1.1 The e500 Processor and the System V ABI

The System V Application Binary Interface, or System V ABI, defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement the interfaces defined in the System V Interface Definition, Issue 3. This includes systems that have implemented UNIX System V Release 4.

The System V Application Binary Interface e500 Processor Supplement (e500 Processor ABI Supplement) described in this document is a supplement to the generic System V ABI and contains information specific to System V implementations built on the e500 Architecture. The generic System V ABI and this supplement together constitute a complete System V Application Binary Interface specification for systems that implement the e500 architecture of the e500 processor family.

In the PowerPC™ architecture, a processor can run in either of two modes: big-endian mode or little-endian mode. (See Section 2.1.2.1, “Byte Ordering.”) Accordingly, this ABI specification defines two binary interfaces, a big-endian ABI and a little-endian ABI. Programs and (in general) data produced by programs that run on an implementation of the big-endian interface are not portable to an implementation of the little-endian interface, and vice versa.

1.2 How to Use the e500 Processor ABI Supplement

Although the generic System V ABI is the prime reference document, this document contains e500-specific and PowerPC architecture-specific implementation details, some of which supersede information in the generic one.

As with the System V ABI, this document refers to other publicly available documents, especially Enhanced PowerPC Architecture, all of which should be considered part of this e500 Processor ABI Supplement and just as binding as the requirements and data it explicitly includes.

1.3 Compatibility with other ABIs

This ABI was constructed with two primary goals:

1. Provide support for the new capabilities of the e500 microarchitecture (64-bit wide registers, SPE data types at the C level, and others)
2. Maintain backwards compatibility where possible, in order to leverage existing tools, libraries, and source code.

Code that avoids hardware support for floating point and AltiVec should be compatible across the ABIs. For example:

- Routines compiled under this ABI can be called by routines compiled under the PowerPC EABI provided all arguments are of integral or pointer type.
- Routines compiled under this ABI can be called by routines compiled under the PowerPC ABI provided all arguments are of integral or pointer type, and the e500 ABI routines do not make use of r2 as an SDATA2 pointer.
- Routines compiled under the PowerPC EABI can be called by routines compiled under this ABI provided all arguments are of integral or pointer type.
- Routines compiled under the PowerPC ABI can be called by routines compiled under this ABI provided all arguments are of integral or pointer type.
- Because the PowerPC EABI has a weaker stack alignment (8 bytes, instead of 16 bytes in both the e500 ABI and the PowerPC ABI), code that links with routines compiled under the PowerPC EABI is no longer e500-ABI-compliant, as the stack may become aligned only on an 8-byte boundary.
- Because the e500 ABI does not pass floating-point parameters in the same manner as either other ABI in most cases, the only floating-point compilation mode that may be compatible across all three ABIs is software floating-point emulation.

1.4 Evolution of the ABI Specification

The System V ABI will evolve over time to address new technology and market requirements, and it will be reissued every three years or so. Each new edition of the specification is likely to contain extensions and additions that will increase the potential capabilities of applications that are written to conform to the ABI.

As with the System V Interface Definition, the System V ABI will implement Level 1 and Level 2 support for its constituent parts. Level 1 support indicates that a portion of the specification will continue to be supported indefinitely. Level 2 support means that a portion of the specification may be withdrawn or altered after the next edition of the System V ABI is made available-that is, a portion of the specification moved to Level 2 support in an edition of the System V ABI specification will remain in effect at least until the following edition of the specification is published.

These Level 1 and Level 2 classifications and qualifications apply to both the generic specification and this supplement. All components of the System V ABI and of this supplement have Level 1 support unless they are explicitly labeled as Level 2.

1.5 Acknowledgements

Motorola would like to recognize the following individuals and companies who devoted substantial time and effort to create this standard:

- Motorola: Brian Grayson, Kumar Gala, Edmar Wienskoski, Kate Stewart, Phil Brownfield, Jerry Young
- Embedded Edge LLC: Dan Malek
- Green Hills Software: Greg Davis, Paul Jensen
- MetaWare: Mark Schimmel
- Metrowerks: Mark Anderson, Laurent Visconti, Bob Campbell
- MontaVista Software: Daniel Jacobowitz, Matt Porter
- QNX Software Systems: Sebastien Marineau, Brian Stecher
- Red Hat: Aldy Hernandez, Richard Henderson
- Wind River: Steve Walk, Brian Nettleton, Paul Beusterien

1.6 References

The following documents may be of interest to the reader of this specification:

- *System V Interface Definition*, Fourth Edition.
- *System V Application Binary Interface*, Edition 4.1.
- *System V Application Binary Interface Draft*, 24 April 2001.
- *System V Application Binary Interface*, PowerPC Processor Supplement, Rev. A.
- *ISO/IEC 9899:1999 Programming Languages - C ("ANSI C99 Spec")*.
- *The PowerPC Architecture: A Specification for A New Family of RISC Processors*. International Business Machines (IBM). San Francisco: Morgan Kaufmann, 1994. (IBM provides a website at http://www.rs6000.ibm.com/resource/technology/ppc-chg.html that documents changes that have been adopted since the 1994 publication.)
- *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* (MPEFPC32B/AD)
- *Motorola's Enhanced PowerPC Architecture Implementation Standards* (working title).
- User's manuals for the various e500 processors.
- *The e500 Programming Interface Manual* (working title).

- *Motorola Book E Implementation Standards: APU ID Reference* (working title)
- *e500 ABI Save/Restore Routines* (working title)
- *DWARF Debugging Information Format*, Version 3 (Revision 2.1, Draft 7), Oct. 29, 2001. Available from http://www.eagercon.com/dwarf/dwarf3std.htm

Chapter 2
Low-Level System Information

2.1 Machine Interface

This section describes processor architecture and data representation.

2.1.1 Processor Architecture

Information about the e500 processor architecture is contained in two different documents:

- *Motorola's Enhanced PowerPC Architecture Implementation Standards* (working title) describes changes to the PowerPC Architecture to suit the embedded market space. Among other things, it describes the concept of application-specific processing units (APUs). Most of the e500 family will contain several APUs to extend their capabilities beyond the standard offering. Where that document and the classic PowerPC Architecture disagree, the e500 complies with Enhanced PowerPC Architecture.
- The appropriate user's manual contains details about the APUs provided on particular processors in the e500 family.

An application program can assume that all instructions defined by the architecture that are not optional exist and work as documented.

To be ABI-conforming, the processor must implement the instructions of the architecture, perform the specified operations, and produce the expected results. The ABI neither places performance constraints on systems nor specifies what instructions must be implemented in hardware. A software emulation of the architecture could conform to the ABI.

Some processors might support the optional instructions in the PowerPC Architecture, or additional non-PowerPC ISA and non-e500 instructions or capabilities. Programs that use those instructions or capabilities do not conform to this e500 ABI; executing them on machines without the additional capabilities gives undefined behavior.

2.1.2 Data Representation

2.1.2.1 Byte Ordering

The architecture defines an 8-bit byte, a 16-bit half word, a 32-bit word, a 64-bit double word, and a 128-bit quadword. Byte ordering defines how the bytes that make up half words, words, double words, and quadwords are ordered in memory. Most significant byte (MSB) byte ordering, or Big-Endian as it is sometimes called, means that the most significant byte is located in the lowest addressed byte position in a storage unit (byte 0). Least significant byte (LSB) byte ordering, or Little-Endian, as it is sometimes called, means that the least significant byte is located in the lowest addressed byte position in a storage unit (byte 0).

The processors that implement the PowerPC architecture support either Big-Endian or Little-Endian byte ordering. This specification defines two ABIs, one for each type of byte ordering. An implementation must state which type of byte ordering it supports. Note that although it is possible on e500 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.

Figure 2-1 through Figure 2-4 show conventions for bit and byte numbering within various width storage units. These conventions apply to both integer data and floating-point data, where the most significant byte of a floating-point value holds the sign and at least the start of the exponent. The figures show Little-Endian byte numbers in the upper right corners, Big-Endian byte numbers in the upper left corners, and bit numbers in the lower corners.

Note that Book E uses 64-bit numbering throughout, including for registers such as the CR that only contain 32 bits. This 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 half words).

NOTE

In the e500 documentation, and in most PowerPC documentation, bits in a word are numbered from left to right (msb to lsb), and figures usually show only the big-endian byte order.

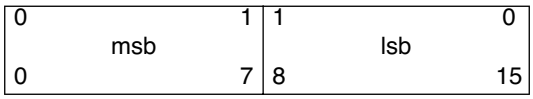

Figure 2-1. Bit and Byte Numbering in Half Words

Figure 2-2 shows bit and byte numbering in words.

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

Figure 2-2. Bit and Byte Numbering in Words

Figure 2-4 shows bit and byte numbering in double words.

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

Figure 2-3. Bit and Byte Numbering in Double Words

Figure 2-4 shows 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

Figure 2-4. Bit and Byte Numbering in Quadwords

2.1.2.2 Fundamental Types

Table 2-1 shows how ANSI C scalar types correspond to those of the e500 processor. For all types, a NULL pointer has the value zero.

Table 2-1. Scalar Types

Type	ANSI C	sizeof	Alignment	e500
Integral	Char	1	Byte	Unsigned byte
	Unsigned char			
	Signed char	1	Byte	Signed byte
	Short	2	Half word	Signed half word
	Signed short			
	Unsigned short	2	Half word	Unsigned half word
	Int	4	Word	Signed word
	Signed int			
	Long int			
	Signed long			
	Enum			
	Unsigned int	4	Word	Unsigned word
	Unsigned long			
	Long long	8	Double word	Signed double word
	Signed long long			
	Unsigned long long	8	Double word	Unsigned double word
Pointer	Any *	4	Word	Unsigned Word
	Any (*) ()			
Floating	Float	4	Word	Single-precision (IEEE)
	Double	8	Double word	Double-precision (IEEE)
	Long Double	16	Quadword	Extended precision (IEEE)

NOTE

Float arguments in general are not required to be promoted to double precision under this ABI, as that would force the use of emulation code on function calls with float arguments. See Section 2.3.4.1, “Float Argument and Return Value Summary” for a summary of when float arguments and return values are promoted.

NOTE

Compilers and systems may implement the double data type in some other way for performance reasons, using a compiler option. Examples of such formats could be two successive floats or even a single float. Such usage does not conform to this ABI, however, and runs the danger of passing a wrongly formatted floating-point number to another, conforming function as an argument. Programs using other formats should transform double floating-point numbers to a conforming format before putting them in permanent storage.

NOTE

Long double support is optional for the e500 ABI.

The expression ‘extended precision (IEEE)’ refers to IEEE 754 double extended precision with a sign bit, a 15-bit exponent with a bias of -16383, and 112 fraction bits (with a leading implicit bit).

Compilers and systems have three choices with respect to long double implementation:

1. Do not provide any long double support. In this case, any use of long doubles should cause a compiler error.
2 Provide ABI-compliant long double support.
3 Provide non-ABI-compliant long double support. Compilers and systems may implement the long double data type in some other way for performance reasons, but must require a compiler option to obtain this behavior. Examples of such formats could be two successive doubles or even a single double. Such usage does not conform to this ABI, however, and runs the danger of passing a wrongly formatted floating-point number to another, conforming function as an argument. Programs using other formats should transform long double floating-point numbers to a conforming format before putting them in permanent storage.

The default behavior of compilers and systems must be one of the first two options; the third option is allowed only in the presence of a compiler option.

NOTE

Even though this ABI describes only software-emulation support for double-precision types, the alignment is the same as if there were hardware support, to minimize differences between this ABI and standard PowerPC ABIs.

NOTE

Because there is no hardware double-precision support, programmers must be careful when writing code with floating-point constants. A statement like "c += 1.0;", where c is a float, causes the compiler to convert c to a double, to insert a call to emulation routines to add that to the constant double 1.0, and then to convert the result back to a float. Compilers for the e500 should likely provide a warning for implicit conversions to double, as they are probably programming errors rather than the desired behavior.

Table 2-3 shows non-ANSI types specified by this ABI.

Table 2-2. Non-ANSI Scalar Types

Type	C type	sizeof	Alignment	e500
SPE (opaque)	`__ev64_opaque__`	8	Double word	Unsigned doubleword

NOTE

The `__ev64_opaque__` type is opaque. The *e500 Programming Interface Manual* describes types to represent a pair of 32-bit values, four 16-bit values, and so forth. For most compiler operations, all of these types can be treated as the opaque 64-bit type `__ev64_opaque__`. With regard to endianness, the representation of bytes within the double word is not defined by this ABI since the type is opaque. The *e500 Programming Interface Manual* describes the proper interpretation of bytes within values for different endianness for all of its types.

For languages like C++ that need to mangle the type name, compilers should use the same mangling as if `__ev64_opaque__` (and any other new types described in the *e500 Programming Interface Manual*) were a user-defined class.

2.1.2.3 Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component; that is, the component with the largest alignment. The size of any object, including aggregates and unions, is always a multiple of the alignment of the object. An array uses the same alignment as its elements. Structure and union objects may require padding to meet size and alignment constraints:

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

In the following examples (Figure 2-5 through Figure 2-10), members' byte offsets for Little-Endian implementations appear in the upper right corners; offsets for Big-Endian implementations in the upper left corners.

```
struct
    { char c;
};

byte aligned, sizeof is 1
```

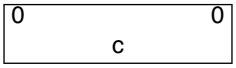

Figure 2-5. Structure Smaller than a Word

Figure 2-6 shows a Little-Endian structure with no padding.

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

word aligned, sizeof is 8
```

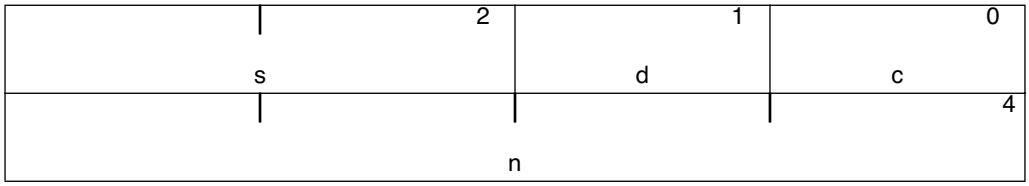

Figure 2-6. No Padding—Little-Endian

Figure 2-7 shows a Big-Endian structure with no padding.

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

word aligned, sizeof is 8
```

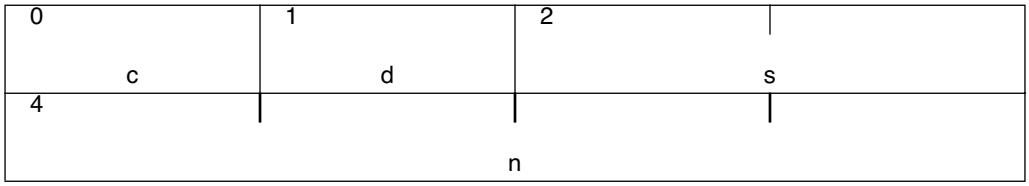

Figure 2-7. No Padding—Big-Endian

Figure 2-8 shows a Little-Endian structure with internal padding.

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

halfword aligned, sizeof is 4
```

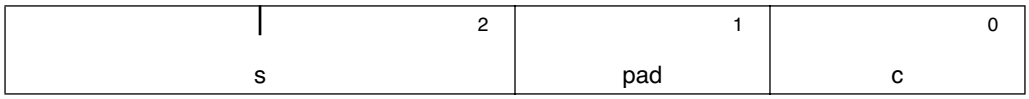

Figure 2-8. Internal Padding—Little-Endian

Figure 2-9 shows a Big-Endian structure with internal padding.

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

halfword aligned, sizeof is 4
```

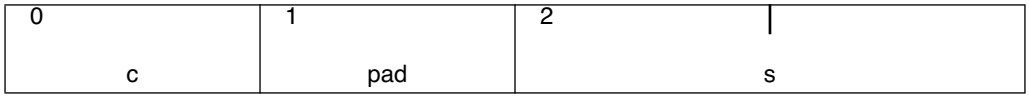

Figure 2-9. Internal Padding—Big-Endian

Figure 2-10 shows a Little-Endian structure with internal and tail padding.

```
struct {
    char c;
    __ev64_opaque__ d;
    short s;
};

doubleword aligned, sizeof is 24
```

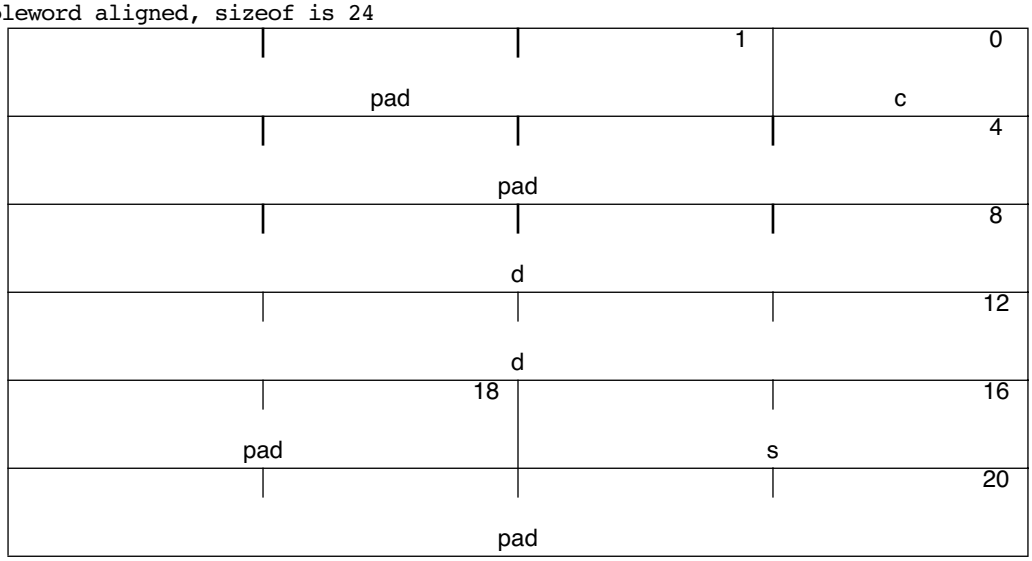

Figure 2-10. Internal and Tail Padding—Little-Endian

Figure 2-11 shows a Big-Endian structure with internal and tail padding.

```
struct {
    char c;
    __ev64_opaque__ d;
    short s;
};

doubleword aligned, sizeof is 24
```

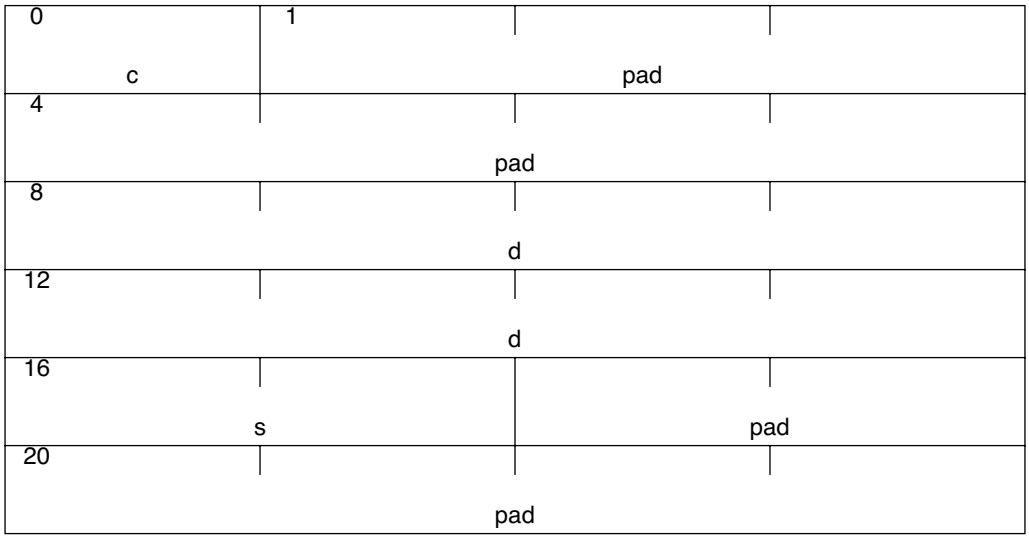

Figure 2-11. Internal and Tail Padding—Big-Endian

Figure 2-12 shows a Little-Endian example of a union allocation.

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

word aligned, sizeof is 4
```

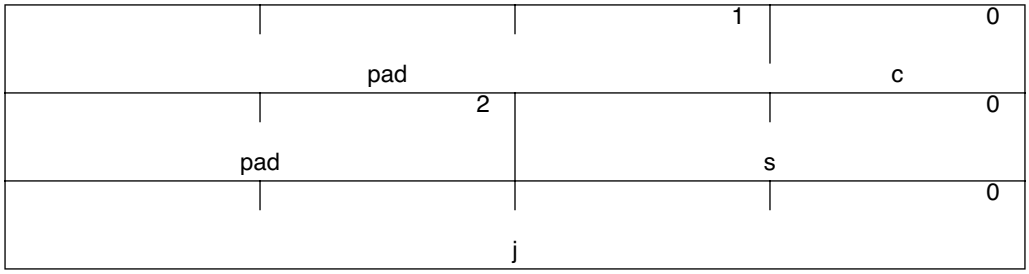

Figure 2-12. Union Allocation—Little-Endian

Figure 2-13 shows a Big-Endian example of a union allocation.

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

word aligned, sizeof is 4
```

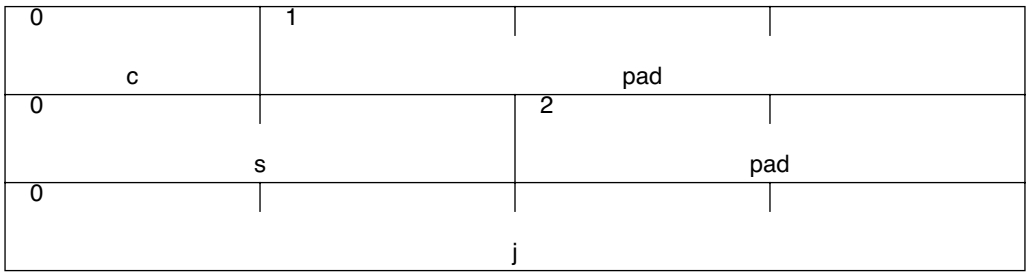

Figure 2-13. Union Allocation—Big-Endian

2.1.2.4 Bit Fields

C struct and union definitions may have bit fields, defining integral objects with a specified number of bits (see Table 2-3).

Table 2-3. Bit Field Ranges

Bit-Field Type	Width w	Range
signed char	1 to 8	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
char	1 to 8	0 to $2^{w} - 1$
unsigned char		
signed short	1 to 16	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
short	1 to 16	0 to $2^{w} - 1$
unsigned short		
signed int	1 to 32	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
signed long		
int	1 to 32	0 to $2^{w} - 1$
enum		
unsigned int		
long		
unsigned long		

Bit fields that are neither signed nor unsigned always have non-negative values. Although they may have type short, int, or long (which can have negative values), bit fields of these types have the same range as bit fields of the same size with the corresponding unsigned type. Bit fields obey the same size and alignment rules as other structure and union members, with the following additions:

- Bit fields are allocated from right to left (least to most significant) on Little-Endian implementations and from left to right (most to least significant) on Big-Endian implementations.
- A bit field must entirely reside in a storage unit appropriate for its declared type. Thus, a bit field never crosses its unit boundary.

- Bit fields must share a storage unit with other structure and union members (either bit field or non-bit field) if and only if there is sufficient space within the storage unit.
- Unnamed bit fields' types do not affect the alignment of a structure or union, although an individual bit field's member offsets obey the alignment constraints. An unnamed, zero-width bit field shall prevent any further member, bit field or other, from residing in the storage unit corresponding to the type of the zero-width bit field. The following examples (Figure 2-14 through Figure 2-24) show struct and union members' byte offsets in the upper right corners for Little-Endian implementations, and in the upper left corners for Big-Endian implementations. Bit numbers appear in the lower corners.

Figure 2-14 shows bit numbering for the value 0x0102_0304.

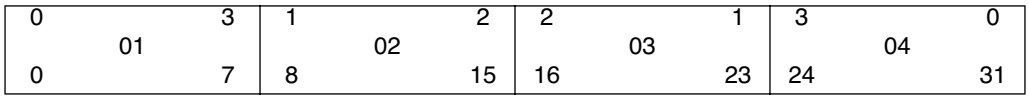

Figure 2-14. Bit Numbering for Value 0x0102_0304

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

word aligned, sizeof is 4
```

Figure 2-15 shows right-to-left (Little-Endian) allocation.

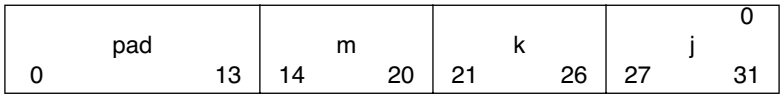

Figure 2-15. Right-to-Left (Little-Endian) Allocation

Figure 2-16 shows left-to-right (Big-Endian) allocation.

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

word aligned, sizeof is 4
```

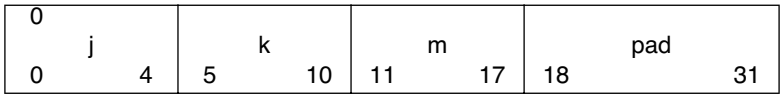

Figure 2-16. Left-to-Right (Big-Endian) Allocation

Figure 2-17 shows Little-Endian boundary alignment.

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

```
word aligned, sizeof is 12
```

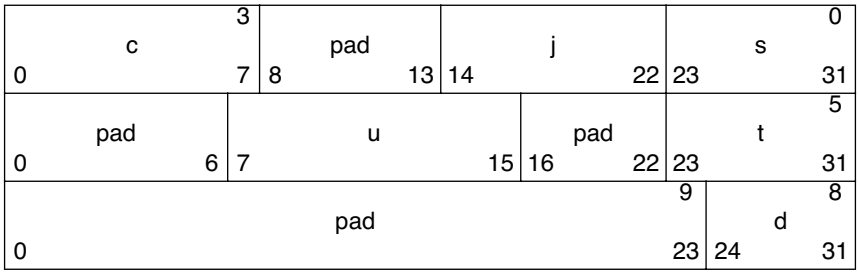

Figure 2-17. Boundary Alignment–Little-Endian

Figure 2-18 shows Big-Endian boundary alignment.

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

word aligned, sizeof is 12
```

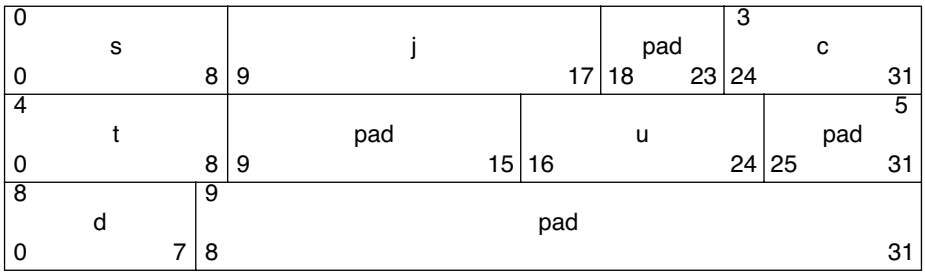

Figure 2-18. Boundary Alignment—Big-Endian

Figure 2-19 shows Little-Endian storage unit sharing.

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

halfword aligned, sizeof is 2
```

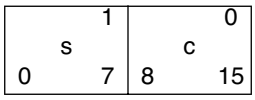

Figure 2-19. Storage Unit Sharing—Little-Endian

Figure 2-20 shows Big-Endian storage unit sharing.

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

halfword aligned, sizeof is 2
```

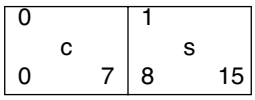

Figure 2-20. Storage Unit Sharing—Big-Endian

Figure 2-21 shows Little-Endian union allocation.

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

halfword aligned, sizeof is 2 little endian:
```

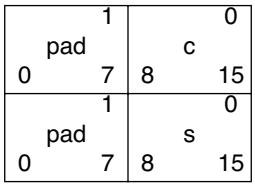

Figure 2-21. Union Allocation—Little-Endian

Figure 2-22 shows Big-Endian union allocation.

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

halfword aligned, sizeof is 2
```

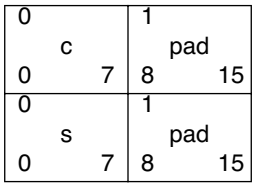

Figure 2-22. Union Allocation—Big-Endian

Figure 2-23 shows Little-Endian unnamed bit fields.

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

byte aligned, sizeof is 9
```

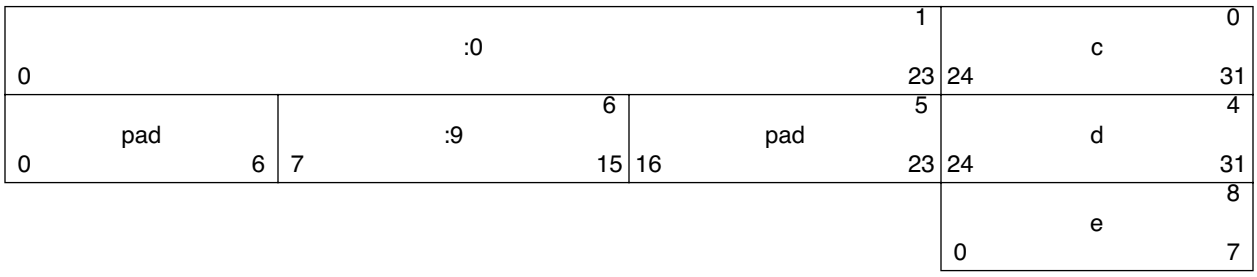

Figure 2-23. Unnamed Bit Fields—Little-Endian

Figure 2-24 shows Big-Endian unnamed bit fields.

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

byte aligned, sizeof is 9
```

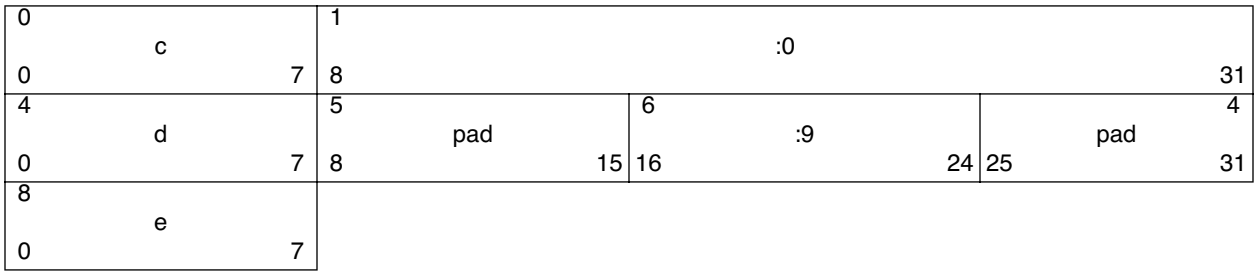

Figure 2-24. Unnamed Bit Fields—Big-Endian

NOTE

In Figure 2-23 and Figure 2-24, the presence of the unnamed int and short fields do not affect the alignment of the structure. They align the named members relative to the beginning of the structure, but the named members may not be aligned in memory on suitable boundaries. For example, the d members in an array of these structures will not all be on an int (4-byte) boundary. As the examples show, int bit fields (including signed and unsigned) pack more densely than smaller base types. The char and short bit fields can be used to force particular alignments, but int is generally more efficient.

2.2 Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, and parameter passing. The system libraries described in Chapter 5, “Libraries,” require this calling sequence.

NOTE

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions as long as they conform to the requirements for stack trace back. Nonetheless, it is recommended that all functions use the standard calling sequences when possible.

NOTE

C programs follow the conventions given here. For specific information on the implementation of C, see Section 2.7, "Coding Examples."

2.2.1 Registers

The e500 architecture provides 32, 32-bit general-purpose registers.

The architecture 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 e500 architecture also provides several special purpose registers. All of the integer and special purpose registers are global to all functions in a running program. Brief register descriptions appear in Table 2-4, followed by more detailed information about the registers.

The volatility of all 64-bit registers is the same for the upper and lower word. Note, however, that if only the lower word is modified by a function, only the lower word need be saved and restored.

Several registers are dedicated (r1 and r13) or reserved (r2). These registers should never be used for any purpose besides their described use.

The SPEFSCR (or EFSCR if only the scalar floating-point instruction set is implemented) is marked as limited-access, which is described in more detail below.

Table 2-4. Processor Registers

Register	Volatility	Usage Name
r0	Volatile	Register which may be modified during function linkage
r1	Dedicated	Stack frame pointer, always valid
r2	Dedicated	Reserved [1]
r3–r4	Volatile	Registers used for parameter passing and return values
r5–r10	Volatile	Registers used for parameter passing
r11–r12	Volatile	Registers that may be modified during function linkage
r13	Dedicated	Small data area pointer register
r14–r31	Nonvolatile	Registers used for local variables
CR0–CR1	Volatile	Condition register fields, each 4 bits wide
CR2–CR4	Nonvolatile	Condition register fields, each 4 bits wide
CR5–CR7	Volatile	Condition register fields, each 4 bits wide
LR	Volatile	Link register
CTR	Volatile	Count register
XER	Volatile	Integer exception register

Table 2-4. Processor Registers (continued)

Register	Volatility	Usage Name
SPEFSCR/ EFSCR [2]	Limited-access	Signal processing and embedded floating-point status and control register/ Embedded floating-point status and control register
ACC	Volatile	SPE accumulator

[1] The default behavior for compilers must be to keep r2 reserved. However, for compatibility with the PowerPC EABI, it is permitted for compilers to support r2 as the sdata2 pointer when the compiler is invoked with an optional flag.

[2] EFSCR if only the scalar floating-point instruction set is implemented.

Register r1 is dedicated to holding the stack pointer.

Registers r14–r31 are nonvolatile; that is, they belong to the calling function. A called function shall save these registers' values before it changes them, restoring their values before it returns.

Registers r0, r3 through r12, and the special-purpose registers LR, CTR, XER, and ACC, as well as the status bits of the SPEFSCR (or EFSCR if only the scalar floating-point instruction set is implemented), are volatile; that is, they are not preserved across function calls. Furthermore, the values in registers r0, r11, and r12 may be altered by cross-module calls, so a function cannot depend on the values in these registers having the same values that were placed in them by the caller.

Register r13 is the small data area pointer. Process startup 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 loader-defined symbol _SDA_BASE_) into r13. Shared objects shall not alter the value in r13. See Section 3.3.1, "Small Data Area (.sdata and .sbss)," for more details.

As in the SVR4 ABI, r2 shall be reserved for system use by default, but compilers may accept a flag to enable compatibility with the EABI. In this case r2, will contain the base, named _SDA2_BASE_, of the ELF sections named .sdata2 and .sbss2, if either section exists in an object file. The base is an address such that every byte in the section is within a signed 16-bit offset of that address. This is analogous to the SVR4 ABI's use of GPR13 to contain _SDA_BASE_, which is the base of sections .sdata and .sbss. A routine in an ELF shared object file shall not use r2. See Section 3.3.2, "Small Data Area 2 (.PPC.EMB.sdata2 and .PPC.EMB.sbss2)," for more details.

Fields CR2, CR3, and CR4 of the condition register are nonvolatile (value on entry must be preserved on exit); the rest are volatile (value in the field need not be preserved).

The SPE APU accumulator register is volatile.

The SPEFSCR (or EFSCR if only the scalar floating-point instruction set is implemented) contains bits with different volatilities. The status bits are volatile (they do not need to be saved and restored), while the rounding mode and exception enable bits are limited-access. Limited-access means that the bits may be changed only by a called function that has the

documented effect of changing them. This is similar to the classic handling of the exception enable and rounding bits in the FPSCR. The *e500 Programming Interface Manual* (PIM) defines the functions that are allowed to change the limited-access SPEFSCR (or EFSCR) bits.

Note that limited-access is different from nonvolatile, as limited-access bits do not need to be saved and restored at function call boundaries. Modifying these bits has a global effect on the application.

The registers in Table 2-5 have assigned roles in the standard calling sequence.

Table 2-5. Register Assignments for Standard Calling Sequence

Register	Description
r1	The stack pointer (stored in r1) shall maintain 16-byte alignment. See Section 2.3, “The Stack Frame,” for details. It shall always point to the lowest allocated, valid stack frame, and grow toward low addresses. The contents of the word at that address always point to the previously allocated stack frame. If required, it can be decremented by the called function; see Section 2.7.8, “Dynamic Stack Space Allocation.”
r3–r10	These sets of volatile registers may be modified across function invocations and shall therefore be presumed by the calling function to be destroyed. They are used for passing parameters to the called function; see Section 2.3.1, “Parameter Passing.” In addition, registers r3 and r4 are used to return values from the called function, as described in Section 2.3.3, “Return Values.”
CR[38]	This bit shall be cleared by the caller of a variable argument function. See Section 2.3.2, “Variable Argument Lists,” for more details.
LR	The link register shall contain the address to which a called function normally returns. LR is volatile across function calls.

Signals can interrupt processes (see signal (BA_OS) in the System V Interface Definition). Functions called during signal handling have no unusual restrictions on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with all registers restored to their original values. Thus, programs and compilers may freely use all registers above except those reserved for system use without the danger of signal handlers inadvertently changing their values.

2.3 The Stack Frame

In addition to the registers, each function may have a stack frame on the runtime stack. This stack grows downward from high addresses. Figure 2-25 shows the stack frame organization. SP in the figure denotes the stack pointer (general purpose register r1) of the called function after it has executed code establishing its stack frame.

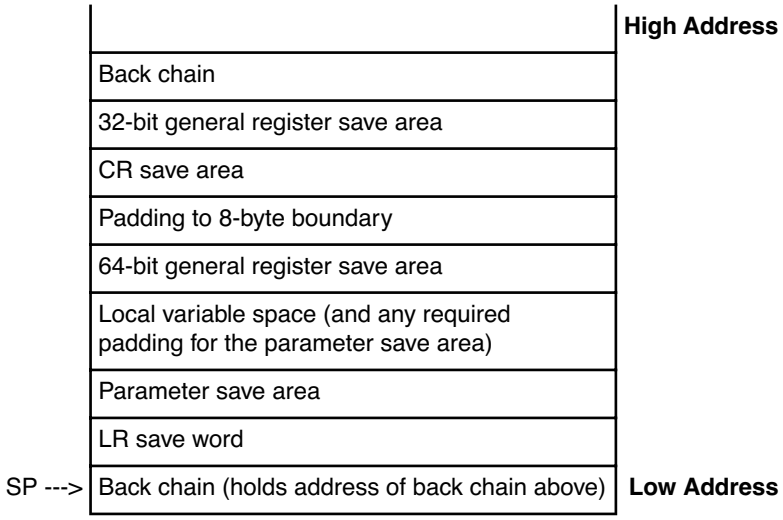

Figure 2-25. Standard Stack Frame

Note that this stack frame layout is different from the standard PowerPC ABI in that the area previously used for holding floating-point registers is unused, and a new area below the CR save area has been created to hold 64-bit values from the general registers.

The following requirements apply to the stack frame:

- The stack pointer shall maintain 16-byte alignment.
- The stack pointer shall point to the first word of the lowest allocated stack frame, the back chain word. The stack shall grow downward, that is, toward lower addresses. The first word of the stack frame shall always point to the previously allocated stack frame (toward higher addresses), except for the first stack frame, which shall have a back chain of 0 (NULL).
- The stack pointer shall be decremented by the called function in its prologue, if required, and restored prior to return.
- The stack pointer shall be decremented and the back chain updated atomically using one of the store word with update instructions, so that the stack pointer always points to the beginning of a linked list of stack frames.
- The parameter list area shall be allocated by the caller and shall be large enough to contain the arguments that the caller stores in it. Its contents are not preserved across calls.
- The sizes of the 32-bit and 64-bit general register save areas may vary within a function and are as determined by the DWARF debugging information.
- Before a function changes the value in the upper word of any nonvolatile general register, rn, it shall save the 64-bit value in rn in the 64-bit general register save area 8*(32-n) bytes below the CR save area (plus any required padding). The 64-bit general save area shall have 8-byte alignment.
- Before a function changes the value in the lower word of any nonvolatile general register, rn, that has not already been saved in the 64-bit general register save area,

it shall save the value in the lower word of rn in the word in the 32-bit general register save area 4*(32-n) bytes before the back chain word of the previous frame.

- Before a function changes the value in any nonvolatile field in the condition register, it shall save the values in all the nonvolatile fields of the condition register at the time of entry to the function in the CR save area. Note that it is sufficient to simply save and restore the entire CR.
- The padding word between the CR save area and the 64-bit general register save area is not needed if there are no registers saved in the 64-bit general register save area and the local variable space does not require 64-bit alignment for its variables.
- Other areas depend on the compiler and the code compiled. The standard calling sequence does not define a maximum stack frame size. The minimum stack frame consists of the first two words, described below, with padding to the required 16-byte alignment. The calling sequence also does not restrict how a language system uses the local variable space of the standard stack frame or how large it should be.

NOTE

This ABI requires 16-byte alignment of the stack to be maintained at all times. Thus, code compiled under this ABI that links with other code compiled under other PowerPC ABIs that only required 8-byte alignment will no longer conform to this ABI, as the stack pointer could end up only 8-byte aligned.

NOTE

The purpose of providing both 32- 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. Also note that, 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 e500 architecture should remain PowerPC EABI compliant (both in regards to stack layout, and in all other ways).

NOTE

In early prototype versions of this ABI, it was permitted for a compiler to choose to save and restore all 64 bits of each modified nonvolatile general register, as long as the debugging information reflects this. However, since this method breaks compatibility with previous ABIs, this method is only permitted for functions that need to save a 64-bit nonvolatile register. Functions that only need to save 32-bit nonvolatiles should emit only 32-bit saves and restores.

The stack frame header consists of the back chain word and the LR save word. The back chain word always contains a pointer to the previously allocated stack frame. Before a function calls another function, it shall save the contents of the link register at the time the function was entered in the LR save word of its caller's stack frame and shall establish its own stack frame.

Except for the stack frame header and any padding necessary to make the entire frame a multiple of 16 bytes in length, a function need not allocate space for the areas that it does not use. If a function does not call any other functions and does not require any of the other parts of the stack frame, it need not establish a stack frame. Any padding of the frame as a whole shall be within the local variable area; the parameter list area shall immediately follow the stack frame header, and the register save areas shall contain no padding except possibly one word of padding between the 32-bit general save area and the 64-bit general save area.

2.3.1 Parameter Passing

For a RISC machine such as the e500, it is generally more efficient to pass arguments to called functions in general registers than to construct an argument list in storage or to push them onto a stack. Since all computations must be performed in registers anyway, memory traffic can be eliminated if the caller can compute arguments into registers and pass them in the same registers to the called function, where the called function can then use them for further computation in the same registers. The number of registers implemented in a processor architecture naturally limits the number of arguments that can be passed in this manner.

For e500, up to eight arguments (words or `__ev64_opaque__` double words) are passed in general purpose registers, loaded sequentially into general purpose registers r3 through r10. If fewer (or no) arguments are passed, the unneeded registers are not loaded and will contain undefined values on entry to the called function.

Note that `__ev64_opaque__` double word arguments to a variable argument function are handled specially; see Section 2.3.2, "Variable Argument Lists," for more details.

Only when worst-case arguments passed from a function do not fit in the eight GPRs provided must a function allocate space for arguments in its stack frame; in that case, it needs to allocate only enough space to hold arguments that do not fit into registers.

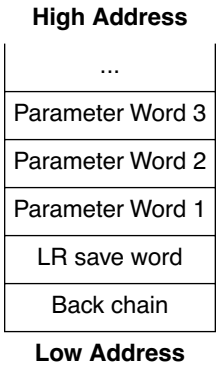

Figure 2-26. Parameter List Area

The following algorithm specifies where argument data is passed for the C language. For this purpose, consider the arguments as ordered from left (first argument) to right, although the order of evaluation of the arguments is unspecified. In this algorithm, gr contains the number of the next available general purpose register, and starg is the address of the next available stack argument word.

- INITIALIZE: Set gr=3, and 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:
- SIMPLE_ARG: A SIMPLE_ARG is one of the following:
  - One of the simple integer types no more than 32 bits wide (char, short, int, long, enum)
  - A single-precision float
  - A DSP 64-bit type (`__ev64_opaque__`)
  - A pointer to an object of any type
  - A struct, union, or long double, any of which shall be treated as a pointer to the object, or to a copy of the object where necessary to enforce call-by-value semantics. Only if the caller can ascertain that the object is constant can it pass a pointer to the object itself.

  If gr>10, go to OTHER. Otherwise, load the argument value into general register gr, set gr to 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.
- LONG_LONG: A LONG_LONG is one of the following:
  - A long long
  - A double
  - an `__ev64_opaque__` being passed to a variable argument function.

  Note that implementations are now required to support a long long data type.

  Also note that doubles are supported only via emulation, and thus will only be passed in two consecutive registers, just like long long types. This preserves

compatibility with the PowerPC EABI when using software floating-point emulation, and allows reuse of legacy emulation routines.

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

Note that even though the general registers can hold 64-bit values, since there are no 64-bit arithmetic operations, long longs are still passed in two consecutive 32-bit general registers, which retains compatibility with the PowerPC EABI.

- OTHER: Arguments not otherwise handled above are passed in the parameter words of the caller's stack frame. Most of the types handled in SIMPLE_ARG, as defined above, 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 (where implemented), double, and `__ev64_opaque__` arguments are considered to have 8-byte size and alignment. Note that float arguments are not required to be promoted to double precision under this ABI (except where mandated by the C language -- see Section 2.3.4.1, "Float Argument and Return Value Summary" for more details),as that would force the use of emulation code on function calls with float arguments.

  If gr>9 and we are handling a LONG_LONG type (long long, double, or `__ev64_opaque__`), then set gr to 11 (to prevent subsequent SIMPLE_ARGs from being placed in registers after LONG_LONG arguments that would no longer fit in the registers). Note that the classic ABI did not specify this step, leaving this situation unclear.

  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, then go to SCAN.

The contents of registers and words skipped by the above algorithm for alignment (padding) are undefined.

As an example, assume the declarations and the function call shown in Figure 2-27. The corresponding register allocation and storage would be as shown in Table 2-6.

```
typedef struct {
    int a, b;
    double dd; /* double word aligned */ }
sparm;
sparm s, t, u;
int c, d; f
float e, f;
double gg, hh, ii;
long double ld;

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

Figure 2-27. Parameter Passing Example

Table 2-6. Parameter Passing Example Register Allocation

General Purpose Registers		Stack Frame Offset	
r3:	c	08:	hh(lo)
r4:	e	0C:	hh(hi)
r5:	d	10:	ptr to t
r6:	ptr to s	14:	(padding)
r7:	f	18:	ii(lo)
r8:	(skipped)	1C:	ii(hi)
r9:	gg(lo) [1]	20:	ptr to u
r10:	gg(hi) [1]	24:	ptr to ld

[1] Note: In Table 2-6, (lo) and (hi) denote the low- and high-addressed word of the double value as stored in memory, regardless of the Endian mode of the implementation. The ptr to arguments are pointers to copies if necessary to preserve call-by-value semantics.

2.3.2 Variable Argument Lists

Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that all arguments are passed on the stack and that arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many implementations. However, they do not work on the PowerPC architecture because some arguments are passed in registers. Portable C programs use the header files <stdarg.h> or <varargs.h> to deal with variable argument lists on processors that implement the PowerPC architecture and other machines as well.

A caller of a function that takes a variable argument list shall clear CR bit 38 (typically used to denote FPR arguments), since no FPR arguments are used. This allows older variable argument functions to be called by e500 ABI functions. The **crclr 6** simplified mnemonic or the **crxor 6,6,6** instruction is recommended for this purpose.

The layout of the parameter save area is 8 consecutive words.

For variable argument functions, `__ev64_opaque__` arguments (both before and after the ellipsis) are passed in the low words of two consecutive registers, in the same manner as long long variables.

For more details on variable-argument handling, see the description of `__va_arg` in Section 5.2.2.2, “Variable-Argument Routine.”

2.3.3 Return Values

Functions shall return values of type int, long, enum, short, char, or a pointer to any type as unsigned or signed integers as appropriate, zero- or sign-extended to 32 bits if necessary, in r3. Functions shall return single-precision float values in r3.

Functions shall return values of 64-bit DSP types (`__ev64_opaque__`) in r3.

A structure or union whose size is less than or equal to 8 bytes shall be returned in r3 and r4, as if it were first stored in an 8-byte aligned memory area and then the low- addressed word were loaded into r3 and the high-addressed word into r4. Bits beyond the last member of the structure or union are not defined.

Values of type long long and unsigned long long, as well as values of type double, shall be returned with the lower addressed word in r3 and the higher in r4.

Values of type long double and structures or unions that do not meet the requirements for being returned in registers are returned in a storage buffer allocated by the caller. The address of this buffer is passed as a hidden argument in r3 as if it were the first argument, causing gr in the argument passing algorithm above to be initialized to 4 instead of 3.

2.3.4 Summary of Float, Double, Short, and Char Argument and Return Value Handling

This section provides a summary of rules described elsewhere concerning the use of single-precision floating point, short, and char data types as arguments and as return values.

2.3.4.1 Float Argument and Return Value Summary

The handling of the float variable type is summarized here.

When using either hardware support (the SPE **efs*** instructions) or software floating-point emulation (SFPE, see Section 5.2.5, “Software Floating-Point Emulation Support Routines”) for the float datatype, float arguments are not promoted to double unless the prototype for the called function specifies a double datatype, the prototype is missing, or the prototype is for a varargs function and the float argument would be passed after the ellipsis.

Table 2-7. Float and Double Argument and Return Value Summary

Function prototype	Promoted?	How arguments are handled if volatile argument registers are available	How arguments are handled if no volatile argument registers are available	How return values are handled
missing	to double	LONG_LONG	OTHER	as double (r3/r4)
double	to double	LONG_LONG	OTHER	as double (r3/r4)
varargs (and after ellipsis)	to double	LONG_LONG	OTHER	n.a.
float	--	SIMPLE_ARG	OTHER	as float

2.3.4.2 Short and Char Argument and Return Value Summary

Note that integer data types shorter than 32 bits (shorts and chars) are sign-extended or zero-extended in all three scalar situations:

- when passed directly in registers
- when passed on the stack
- when returned in registers

2.3.5 Stack Frame Examples

This section describes several possible functions, and shows their respective stack frame layouts.

2.3.5.1 Simple Function

If a function does not need to save any nonvolatile registers (GPRs or the CR), then the size of its stack frame is determined by its local variable usage and parameter save area.

2.3.5.1.1 Minimal Stack Frame: No Local Variables or Saved Parameters

If there are no local variables or saved parameters, then the contents of the stack frame are two words of padding and the stack frame header (saved LR and stack pointer), as shown in Table 2-8.

Table 2-8. Minimal Stack Frame

Address Offset from Previous Stack Frame	Address Offset from New Stack Frame	Description
0x0	0x10	back chain (16-byte aligned)
-0x4, -0x8	0x8,0xC	2 words of padding
-0xc	0x4	LR
-0x10	0x0	new back chain

2.3.5.1.2 Local Variables or Saved Parameters Only

If there are only local variables and/or saved parameters in the stack frame, then padding is needed in the local variable space to keep the stack frame aligned to a 16-byte boundary.

Padding may also be needed within both the parameter save area and the local variable space. For example, consider a non-varargs function that saves a 32-bit parameter at the bottom of the parameter save area, followed by an `__ev64_opaque__` parameter, with no local variables. The stack frame is shown in Table 2-9.

Table 2-9. Padding in Both Parameter Save Area and in Local Variable Space

Address Offset from Previous Stack Frame	Address Offset from New Stack Frame	Description
0x0	0x20	back chain (16-byte aligned)
-0x4, -0x8	0x18, 0x1c	2 words of padding (for stack frame)
-0x10	0x10	second parameter (64 bits)
-0x14	0xc	padding (for second parameter)
-0x18	0x8	first parameter (32 bits)
-0x1c	0x4	LR
-0x20	0x0	new back chain

2.3.5.2 Functions that Save Nonvolatile Registers

There are three different kinds of nonvolatile registers that may be saved in a stack frame: 32-bit general purpose registers, 64-bit general purpose registers, and the CR. Since the 32-bit general register save area and the CR save word are contiguous, the padding for the upper word depends on the total size of the 32-bit general purpose register save area combined with the CR save word.

2.3.5.2.1 Function with No 64-Bit Nonvolatile Usage

If no 64-bit nonvolatile registers need to be saved in the stack frame, then there is no 64-bit general register save area, and there will never be a word of padding below the CR save area purely to align the nonexistent 64-bit general register save area to an 8-byte boundary. Note that the local variable space may require padding to maintain proper alignment for its variables.

Consider a function that saves 5 nonvolatile 32-bit registers, and has no local variable space or parameter save area. The stack layout for this function is shown in Table 2-10.

Table 2-10. 32-Bit Nonvolatile Example

Address Offset from Previous Stack Frame	Address Offset from New Stack Frame	Description
0x0	0x20	back chain (16-byte aligned)
-0x4	0x1c	r31 (32 bits)
-0x8	0x18	r30 (32 bits)
-0xc	0x14	r29 (32 bits)
-0x10	0x10	r28 (32 bits)
-0x14	0xc	r27 (32 bits)
-0x18	0x8	1 word of padding

Table 2-10. 32-Bit Nonvolatile Example (continued)

Address Offset from Previous Stack Frame	Address Offset from New Stack Frame	Description
-0x1c	0x4	LR
-0x20	0x0	new back chain

2.3.5.2.2 Function with Both 32-Bit and 64-Bit Nonvolatile Usage

If a function saves both 32-bit and 64-bit nonvolatile registers on the stack, padding may be required in two places: directly below the CR save word (or the 32-bit general save area if the CR save word is not needed), and in the local variable space.

Consider a function that saves 5 nonvolatile 32-bit registers and 3 64-bit nonvolatile registers. The stack layout for this function is shown in Table 2-11.

Table 2-11. 32-Bit and 64-Bit Nonvolatile Example

Address Offset from Previous Stack Frame	Address Offset from New Stack Frame	Description
0x0	0x40	back chain (16-byte aligned)
-0x4	0x3c	r31 (32 bits)
-0x8	0x38	r30 (32 bits)
-0xc	0x34	r29 (32 bits)
-0x10	0x30	r28 (32 bits)
-0x14	0x2c	r27 (32 bits)
-0x18	0x28	1 word of padding
-0x20	0x20	r26 (64 bits)
-0x28	0x18	r25 (64 bits)
-0x30	0x10	r24 (64 bits)
-0x38,-0x34	0x8,0xc	two words of padding
-0x3c	0x4	LR
-0x40	0x0	new back chain

2.3.5.3 Maximum Amount of Stack Frame Padding

Note that even with two different padding areas, the total padding within a stack frame due to nonvolatile usage (i.e., not counting internal padding for 64-bit variables in the local variable space and the parameter save area) will never be more than 3 words: If there are no 64-bit nonvolatiles saved, then all of the padding for the frame is contiguous, and is either 0, 1, 2, or 3 words; if there are 64-bit nonvolatiles saved, there is either 0 or 1 words of padding above the 64-bit general register save area, and since the 64-bit general register

save area is 64-bit aligned, there is either 0 or 2 words of padding below it in the local variable space to guarantee 16-byte alignment for the entire stack frame.

2.4 Operating System Interface—Optional

This section is optional. No specifications in this section are required for ABI compliance.

2.4.1 Virtual Address Space

Processes execute in a 32-bit virtual address space. Memory management translates virtual addresses to physical addresses, hiding physical addressing and letting a process run anywhere in the system's real memory. Processes typically begin with three logical segments: text, data, and stack. An object file may contain more segments (for example, for debugger use), and a process can also create additional segments for itself with system services.

NOTE

The term 'virtual address' as used in this document refers to a 32-bit address generated by a program, as contrasted with the physical address to which it is mapped. The PowerPC Architecture documentation refers to this type of address as an effective address.

2.4.2 Page Size

Memory is organized into pages, which are the system's smallest units of memory allocation. Book E allows processors to support multiple page sizes. Processes may call sysconf(BA_OS) to determine the system's current page size. The e500 supports variable page sizes. This ABI assumes a default minimum page size of 4096 bytes (4 Kbytes), but allows the underlying operating system to cluster pages into larger logical power-of-two page sizes.

2.4.3 Virtual Address Assignments

Conceptually, processes have the full 32-bit address space available to them. In practice, however, several factors limit the size of a process:

- The system reserves a configuration-dependent amount of virtual space.
- A tunable configuration parameter limits process size.
- A process whose size exceeds the system's available combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical

memory and secondary storage are shared resources. System load, which can vary from one program execution to the next, affects the available amounts.

Figure 2-28 shows the virtual address configuration on the PowerPC Architecture. The segments with different properties are typically grouped in different areas of the address space. A reserved area resides at the top of the virtual space and is used by the system. The loadable segments may begin at zero (0); the exact addresses depend on the executable file format (see Chapter 3, "Object Files," and Chapter 5, "Libraries."). The process' stack and dynamic segments reside below the system-reserved area. Processes can control the amount of virtual memory allotted for stack space, as described below.

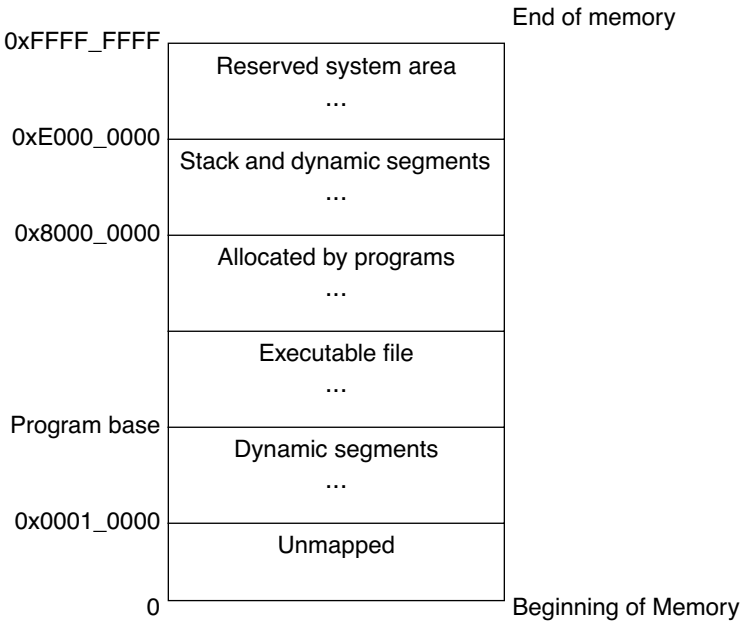

Figure 2-28. Virtual Address Configuration

NOTE

Although application programs may begin at virtual address 0, they conventionally begin above 0x1_0000 (64 Kbytes), leaving the initial 64 Kbytes with an invalid address mapping. Processes that reference this invalid memory (for example, by dereferencing a null pointer) generate an access exception trap, as described in the section Exception Interface in this chapter.

NOTE

A program base of 0x1000_0000 (32 Mbytes) is recommended, for reasons given in Chapter 4, "Program Loading and Dynamic Linking." This implies the first valid instructions starting around 0x1000_0400 or later, to provide room for the ELF header.

As Figure 2-28 shows, the system reserves the high end of virtual space, with a process' stack and dynamic segments below that. Although the exact boundary between the reserved area and a process depends on the system's configuration, the reserved area shall not consume more than 512 Mbytes from the virtual address space. Thus, the user virtual address range has a minimum upper bound of 0xDFFF_FFFF. Individual systems may reserve less space, increasing the process virtual memory range. More information follows in Section 2.4.4, "Managing the Process Stack."

Although applications may control their memory assignments, the typical arrangement follows the diagram above. When applications let the system choose addresses for dynamic segments (including shared object segments), it will prefer addresses below the beginning of the executable and above 64 Kbytes, or addresses above 2 Gbytes. This leaves the middle of the address spectrum, those addresses above the executable and below 2 Gbytes, available for dynamic memory allocation with facilities such as malloc(BA_OS).

2.4.4 Managing the Process Stack

The section Process Initialization in this chapter describes the initial stack contents. Stack addresses can change from one system to the next-even from one process execution to the next on a single system. A program, therefore, should not depend on finding its stack at a particular virtual address.

A tunable configuration parameter controls the system maximum stack size. A process can also use setrlimit(BA_OS) to set its own maximum stack size, up to the system limit. The stack segment is both readable and writable.

2.4.5 Coding Guidelines

Operating system facilities, such as mmap(KE_OS), allow a process to establish address mappings in two ways. First, the program can let the system choose an address. Second, the program can request the system to use an address the program supplies. The second alternative can cause application portability problems because the requested address might not always be available. Differences in virtual address space can be particularly troublesome between different architectures, but the same problems can arise within a single architecture.

Processes' address spaces typically have three segments that can change size from one execution to the next: the stack [through setrlimit(BA_OS)]; the data segment [through malloc(BA_OS)]; and the dynamic segment area [through mmap(KE_OS)]. Changes in one area may affect the virtual addresses available for another. Consequently, an address that is available in one process execution might not be available in the next. Thus, a program that used mmap(KE_OS) to request a mapping at a specific address could appear to work in some environments and fail in others. For this reason, programs that want to establish a mapping in their address space should let the system choose the address.

Despite these warnings about requesting specific addresses, the facility can be used properly. For example, a multiple-process application might map several files into the address space of each process and build relative pointers among the files' data. This could be done by having each process ask for a certain amount of memory at an address chosen by the system. After each process receives its own private address from the system, it would map the desired files into memory, at specific addresses within the original area. This collection of mappings could be at different addresses in each process but their relative positions would be fixed. Without the ability to ask for specific addresses, the application could not build shared data structures because the relative positions for files in each process would be unpredictable.

2.4.6 Processor Execution Modes

Two execution modes exist in the PowerPC Architecture: user and supervisor. Typical processes run in user mode (the less privileged mode). The operating system kernel runs in supervisor mode. A program executes an **sc** instruction to change to supervisor mode.

Note that the ABI does not define the implementation of individual system calls. Instead, programs shall use the system libraries described in Chapter 5, "Libraries."

2.5 Exception Interface—Optional

This section is optional. No specifications in this section are required for ABI compliance.

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, 1) information (such as the address of the instruction that should be executed after control is returned to the original program and the contents of the machine state register) is saved, 2) program control passes from user to supervisor level, and 3) software continues execution at an address (exception vector) predetermined for each exception.

Exceptions may be synchronous or asynchronous. Synchronous exceptions, which are caused by instruction execution, can be explicitly generated by a process. The operating system handles an exception either by completing the faulting operation in a manner transparent to the application or by delivering a signal to the application. The correspondence between exceptions and signals is shown in Table 2-12.

Table 2-12. Exceptions and Signals

Exception	Name	Signal Examples
Illegal instruction	SIGILL	Illegal or privileged instruction Invalid instruction form Optional, unimplemented instruction
Storage access	SIGSEGV	Unmapped instruction or data location access Storage protection violation
Alignment	SIGBUS	Invalid data item alignment
Trap instruction	SIGTRAP	Execution of tw instruction (see Note below)
Floating unavailable	SIGFPE	Floating instruction is not implemented
Floating exception	SIGFPE	Floating-point overflow or underflow Floating-point divide by zero Floating-point conversion overflow Other enabled floating-point exceptions
SPE exceptions	SIGILL	SPE APU is not enabled Enabled SPE vector floating-point exceptions
Cache-locking overlock	SIGSEGV	Cache-locking DSI or ISI exception: all ways already locked.

NOTE

The **tw** instruction with all five condition bits set is reserved for system use (for example, breakpoint implementation), so applications should not rely on the behavior of such traps.

The signals that an exception may give rise to are SIGILL, SIGSEGV, SIGBUS, SIGTRAP, and SIGFPE. If one of these signals is generated due to an exception when the signal is blocked, the behavior is undefined.

Due to the pipelined nature of the processors that implement the PowerPC architecture, more than one instruction may be executing concurrently. When an exception occurs, all unexecuted instructions that appear earlier in the instruction stream are allowed to complete. As a result of completing these instructions, additional exceptions may be generated. All such exceptions are handled in order.

The operating system partitions the set of concurrent exceptions into subsets, all of whose exceptions share the same signal number. Each subset of exceptions is delivered as a single signal. The multiple signals resulting from multiple concurrent exceptions are delivered in unspecified order.

2.6 Process Initialization—Optional

This section is optional. No specifications in this section are required for ABI compliance.

This section describes the machine state that exec(BA_OS) creates for so-called infant processes, including argument passing, register usage, and stack frame layout.

Programming language systems use this initial program state to establish a standard environment for their application programs. For example, a C program begins executing at a function named main, conventionally declared in the way described in Figure 2-29.

```
extern int main (int argc, char *argv[], char *envp[]);
```

Figure 2-29. Declaration for Main

Briefly, argc is a non-negative argument count; argv is an array of argument strings, with argv[argc] == 0; and envp is an array of environment strings, also terminated by a NULL pointer.

Although this section does not describe C program initialization, it gives the information necessary to implement the call to main or to the entry point for a program in any other language.

2.6.1 Registers

When a process is first entered (from an exec(BA_OS) system call), the contents of registers other than those listed below are unspecified. Consequently, a program that requires registers to have specific values must set them explicitly during process initialization. It should not rely on the operating system to clear all registers. Table 2-13 lists registers whose contents are specified.

Table 2-13. Registers with Specified Contents

Register	Description
r1	The initial stack pointer, aligned to a 16-byte boundary and pointing to a word containing a NULL pointer.
r3	Contains argc, the number of arguments.
r4	Contains argv, a pointer to the array of argument pointers in the stack. The array is immediately followed by a NULL pointer. If there are no arguments, r4 points to a NULL pointer.
r5	Contains envp, a pointer to the array of environment pointers in the stack. The array is immediately followed by a NULL pointer. If no environment exists, r5 points to a NULL pointer.
r6	Contains a pointer to the auxiliary vector. The auxiliary vector shall have at least one member, a terminating entry with an a_type of AT_NULL (see Figure 2-30 and Table 2-14).
r7	Contains a termination function pointer. If r7 contains a nonzero value, the value represents a function pointer that the application should register with atexit(BA_OS). If r7 contains zero, no action is required.
SPEFSCR /EFSCR [1]	Contains 0, specifying round-to-nearest mode, cleared status bits, and the disabling of floating-point exceptions for both the upper and lower halves.

[1] EFSCR if only the scalar floating-point instruction set is implemented

2.6.2 Process Stack

Every process has a stack, but the system defines no fixed stack address. Furthermore, a program's stack address can change from one system to another-even from one process invocation to another. Thus the process initialization code must use the stack address in

general purpose register r1. Data in the stack segment at addresses below the stack pointer contain undefined values.

Whereas the argument and environment vectors transmit information from one application program to another, the auxiliary vector conveys information from the operating system to the program. This vector is an array of structures, which are defined in Figure 2-30.

```
typedef struct {
        int     a_type;
        union {
            long a_val;
        void *a_ptr;
        void (*a_fcn)();
    } a_un;
} auxv_t;
```

Figure 2-30. Auxiliary Vector Structure

The structures are interpreted according to the a_type member, as shown in Table 2-14.

Table 2-14. Auxiliary Vector Types, a_type

Name	Value	a_un
AT_NULL	0	Ignored
AT_IGNORE	1	Ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr
AT_DCACHEBSIZE	10	a_val
AT_ICACHEBSIZE	11	a_val
AT_UCACHEBSIZE	12	a_val

a_type auxiliary vector types are described in Table 2-15.

Table 2-15. a_type Auxiliary Vector Types

Name	Description
AT_NULL	The auxiliary vector has no fixed length; instead an entry of this type denotes the end of the vector. The corresponding value of a_un is undefined.
AT_IGNORE	This type indicates the entry has no meaning. The corresponding value of a_un is undefined.
AT_EXECFD	As Chapter 5 in the System V ABI describes, exec(BA_OS) may pass control to an interpreter program. When this happens, the system places either an entry of type AT_EXECFD or one of type AT_PHDR in the auxiliary vector. The entry for type AT_EXECFD uses the a_val member to contain a file descriptor open to read the application program's object file.
AT_PHDR	Under some conditions, the system creates the memory image of the application program before passing control to an interpreter program. When this happens, the a_ptr member of the AT_PHDR entry tells the interpreter where to find the program header table in the memory image. If the AT_PHDR entry is present, entries of types AT_PHENT, AT_PHNUM, and AT_ENTRY must also be present. See the section Program Header in Chapter 5 of the System V ABI and the section Section 4.1, "Program Loading—Extended Conformance," of this processor supplement for more information about the program header table.
AT_PHENT	The a_val member of this entry holds the size, in bytes, of one entry in the program header table to which the AT_PHDR entry points.
AT_PHNUM	The a_val member of this entry holds the number of entries in the program header table to which the AT_PHDR entry points.
AT_PAGESZ	If present, this entry's a_val member gives the system page size in bytes. The same information is also available through sysconf(BA_OS).
AT_BASE	The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory. See the section Program Header in Chapter 5 of the System V ABI for more information about the base address.
AT_FLAGS	If present, the a_val member of this entry holds 1-bit flags. Bits with undefined semantics are set to zero.
AT_ENTRY	The a_ptr member of this entry holds the entry point of the application program to which the interpreter program should transfer control.
AT_DCACHEBSIZE	The a_val member of this entry gives the data cache block size for processors on the system on which this program is running. If the processors have unified caches, AT_DCACHEBSIZE is the same as AT_UCACHEBSIZE.
AT_ICACHEBSIZE	The a_val member of this entry gives the instruction cache block size for processors on the system on which this program is running. If the processors have unified caches, AT_DCACHEBSIZE is the same as AT_UCACHEBSIZE.
AT_UCACHEBSIZE	The a_val member of this entry is zero if the processors on the system on which this program is running do not have a unified instruction and data cache. Otherwise, it gives the cache block size.

Other auxiliary vector types are reserved. No flags are currently defined for AT_FLAGS on the PowerPC Architecture. When a process receives control, its stack holds the arguments, environment, and auxiliary vector from exec(BA_OS). Argument strings, environment strings, and the auxiliary information appear in no specific order within the information block; the system makes no guarantees about their relative arrangement. The system may also leave an unspecified amount of memory between the null auxiliary vector entry and the

beginning of the information block. The back chain word of the first stack frame contains a null pointer (0). A sample initial stack is shown in Figure 2-31.

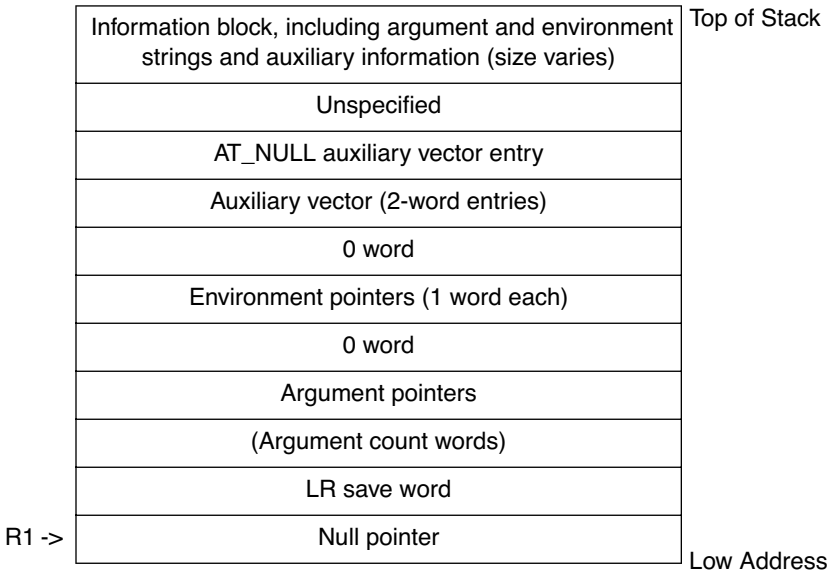

Figure 2-31. Initial Process Stack

2.7 Coding Examples

This section describes example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections discussed how a program may use the machine or the operating system, and they specified what a program may and may not assume about the execution environment. Unlike previous material, the information in this section illustrates how operations may be done, not how they must be done.

As before, examples use the ANSI C language. Other programming languages may use the same conventions displayed below, but failure to do so does not prevent a program from conforming to the ABI. Two main object code models are available:

- Absolute code. Instructions can hold absolute addresses under this model. To execute properly, the program must be loaded at a specific virtual address, making the program's absolute addresses coincide with the process' virtual addresses.
- Position-independent code. Instructions under this model hold relative addresses, not absolute addresses. Consequently, the code is not tied to a specific load address, allowing it to execute properly at various positions in virtual memory.

The following sections describe the differences between these models. When different, code sequences for the models appear together for easier comparison.

NOTE

The examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences or to reproduce compiler output. None of them reference data in the small data area.

2.7.1 Code Model Overview

When the system creates a process image, the executable file portion of the process has fixed addresses and the system chooses shared object library virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared objects conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without having to change segment images. Thus multiple processes can share a single shared object text segment, even if the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques:

- Control transfer instructions hold addresses relative to the effective address (EA) or use registers that hold the transfer address. An EA-relative branch computes its destination address in terms of the current EA, not relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in instructions (in the text segment), the compiler generates code to calculate an absolute address (in a register or in the stack or data segment) during execution.

Because the PowerPC Architecture provides EA-relative branch instructions and also branch instructions using registers that hold the transfer address, compilers can satisfy the first condition easily.

A global offset table (GOT) provides information for address calculation. Position-independent object files (executable and shared object files) have a table in their data segment that holds addresses. When the system creates the memory image for an object file, the table entries are relocated to reflect the absolute virtual address as assigned for an individual process. Because data segments are private for each process, the table entries can change -- unlike text segments, which multiple processes share.

Two position-independent models give programs a choice between more efficient code with some size restrictions and less efficient code without those restrictions. Because of the processor's architecture, a global offset table with no more than 16,384 entries (65,536 bytes) is more efficient than a larger one. Programs that need more entries must use the larger, more general code. In the following sections, the term 'small model' position-independent code is used to refer to code that assumes the smaller global offset table, and large model position-independent code is used to refer to the general code.

2.7.2 Function Prologue and Epilogue

This section describes functions' prologue and epilogue code. A function's prologue establishes a stack frame, if necessary, and may save any nonvolatile registers it uses. A function's epilogue generally restores registers that were saved in the prologue code, restores the previous stack frame, and returns to the caller.

Except for the rules below, this ABI does not mandate predetermined code sequences for function prologues and epilogues. However, the following rules, which permit reliable call chain backtracing, shall be followed:

1. 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 word of its caller's frame.
2. If a function establishes a stack frame, it shall update the back chain word of the stack frame atomically with the stack pointer (r1) using one of the store word with update instructions.
   - For small (no larger than 32 Kbytes) stack frames, this may be accomplished with a Store Word with Update instruction with an appropriate negative displacement.
   - For larger stack frames, the prologue shall load a volatile register with the two's complement of the size of the frame (computed with **addis** and **addi** or **ori** instructions) and issue a Store Word with Update Indexed instruction.
3. The only permitted references with negative offsets from the stack pointer are those described here for establishing a stack frame.
4. When a function deallocates its stack frame, it must do so atomically, either by loading the stack pointer (r1) with the value in the back chain field or by incrementing the stack pointer by the same amount by which it has been decremented.

In-line code may be used to save or restore nonvolatile general registers that the function uses. However, if there are many registers to be saved or restored, it may be more efficient to call one of the system subroutines described below.

Unlike some other processors that implement the PowerPC architecture, the e500 supports load and store multiple PowerPC 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.

If any of the nonvolatile fields of the Condition Register (CR) are used, they must also be preserved and restored. On several implementations, performing the CR restore using several single-field **mtcrf** instructions is more efficient than a single multi-field **mtcrf**.

A function that is position independent will probably want to load a pointer to the global offset table into a nonvolatile register. This may be omitted if the function makes no

external data references. If external data references are only made within conditional code, loading the global offset table pointer may be deferred until it is known to be needed.

2.7.3 Register Saving and Restoring Functions

2.7.3.1 Background

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.

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.

There are four cases to consider with respect to saving/restoring nonvolatiles for a function:

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

Saving and restoring functions also have variants (_g for saves, _x and _t for restores) that bundle some common prologue and epilogue operations to reduce overhead and code footprint by a few instructions. These are discussed in more detail below.

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

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 to determine how many registers to save.

2.7.3.2 Calling Conventions

The register saving and restoring functions described in this section use nonstandard calling conventions which require them to be statically linked into any executable or shared object modules in which they are used. Thus their interfaces are private, within module interfaces, and therefore are not part of the ABI. They are defined here only to encourage uniformity among compilers in the code used to save and restore registers.

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

On entry, all the 32-bit save/restore functions described in this section expect r11 to contain the address of the word just beyond the end of the 32-bit general register save area (the back chain word), and they leave r11 undisturbed. The value held in r11 for the 64-bit save/restore functions varies on the type of function.

- All the non-CTR 64-bit save/restore functions described in this section expect r11 to contain the address of the back chain word, adjusted by subtracting 144 (0x90). 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/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 double word 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/restored and adding 8*13 = 104 (0x88). These two adjustments allow positive offsets, and adjust so that the last register saved ends up directly below the 32-bit general register save area. Note that these adjustments allow a single routine, with fixed offsets, to be used across all potential cases. Note that the double word beyond the 64-bit general save area could be the low word of the 32-bit general 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.

These rules are summarized in Table 2-16.

Table 2-16. r11 Contents at Entry

Function Type	r11 Contents
save/restore 32-bit values (rM–r31)	address of back chain
save/restore 64-bit values (rM–r31)	address of back chain (or pad word below CR save word, if CR is saved) -0x90
save/restore 64-bit values (rM–rN, where N != 31)	address of low end of 32-bit save area/CR save word/padding, adjusted by subtracting 8*N and adding 0x58.

For example, assume a stack frame as described in Figure 2-25, with the back chain word at address 0x400, and where the CR does not need to be saved. To save only 32-bit values, r11 should contain 0x400 on entry to the 32-bit register save routine. To save only 64-bit values, r11 should contain 0x400 - 0x90 = 0x370. Figure 2-32 shows an example of saving some 32-bit and some 64-bit values.
Figure 2-32 shows the standard stack frame.

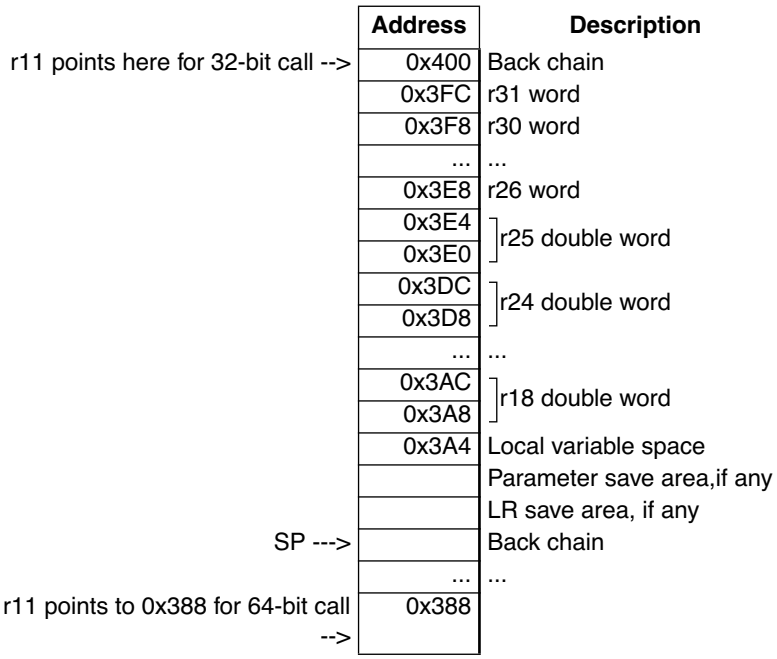

Figure 2-32. Standard Stack Frame

To save r26–r31 as 32-bit values, as shown in the stack frame in Figure 2-32, and r18–r25 as 64-bit values, r11 should contain 0x400 (address of the back chain word) on entry to the appropriate 32-bit save routine (_save32gpr_26), and then be adjusted to contain 1000 - 8*25 + 104 = 904 (0x3e8 - 8 * 25 + 0x68 = 0x388) in preparation for the call to the 64-bit save routine (_save64gpr_ctr_18). r25 will be stored at 0x388+0x58 = byte 0x3e0, which is the double word directly preceding the 32-bit save area. Note that the r11 pointer for the 64-bit call does not necessarily point to any particularly meaningful location directly; it points to an address such that the fixed offsets make the register saves and restores line up appropriately. (In fact, r11 points to where r14 would be stored if it needed saving.)

2.7.3.3 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 register affected. The function names below use the notation [32/64] to designate the use of a 32 for the 32-bit general register functions and a 64 for the 64-bit general 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, one should call _save32gpr_18.

Figure 2-33 shows an example implementation.

There are two families of register saving functions:

- The following simple register saving functions save the indicated registers and return:
  - _save32gpr_m
  - _save64gpr_m and _save64gpr_ctr_m
- The following GOT register saving functions do not return directly
  - _save32gpr_m_g
  - _save64gpr_m_g and _save64gpr_ctr_m_g

  Instead they branch to _GLOBAL_OFFSET_TABLE_-4, relying on a **blrl** instruction at that address (see Section 4.3.2, “Global Offset Table”) to return to the caller of the save function with the address of the 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:
  - _rest32gpr_m
  - _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 back chain word, restore the link register from the LR save word, remove the stack frame, and return through the link register:
  - _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:
  - _rest32gpr_m_t
  - _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 sees an apparent call from the function above the one that made the tail call and, when done, returns directly to it.

For example, the following code implements a tail call to the routine tail:

```
function:
        ...
        bl   _rest32gpr_25_t
        b    tail
```

Note that there are no functions _rest64gpr_ctr_m_x or _reset64gpr_ctr_m_t, as the back chain word is not directly above the location of the 64-bit save area in these cases. In this case, the 64-bit registers should be restored first, followed by a call to _rest32gpr_m_x or _rest32gpr_m_t.

Note also that if a CR save word is used, even if only 64-bit registers are saved, _rest64gpr_m_x and rest64gpr_m_t can not be used, as the back chain word is not directly above the end of the 64-bit save area.

Figure 2-33 shows sample implementations of several of these functions. These sample implementations are also available as “e500 ABI Save/Restore Routines.”

simple save routines:

```
_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
```

simple restore routines:

```
_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
```

Figure 2-33. Implementations of Several of the Save/Restore Routines

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) do the following in replace of the blr:

`_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 are similar to the exit functions, except they skip the **mtlr** instruction.

Note that the CTR-based 64-bit restore functions can not perform the exit and tail optimizations as implemented here, as the address of the back chain word and the return address are not at a fixed offset from r11.

Note that for slightly higher performance in the restore variants, the lwz of r0 and the restore of r31 could be reordered (but the label for _rest[32/64]gpr_31* must now point to the lwz of r0, not the load of r31). Here is an example 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
```

Figure 2-34 shows sample prologue and epilogue code with full saves of all the nonvolatile general registers (r14 through r25 as 64-bit, r26 through r31 as 32-bit) and a stack frame size of less than 32 Kbytes. 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 and does no dynamic stack allocation.

NOTE

This code assumes that the size of the module (executable or shared object) in which the code appears is such that a relative branch is able to reach from any part of the text section to any part of the global offset table (or the procedure linkage table, discussed in Chapter 4, "Program Loading and Dynamic Linking."). Because relative branches can reach +/- 32 Mbytes, this is not considered a serious restriction.

Figure 2-34 shows prologue and epilogue sample code.

```
function:
       mflr    r0                      # Save return address in caller's frame
       stw     r0,4(r1)                # . . .
       li      r0,12                   # Set up CTR with number of 64-bit regs to save.
       mr      r11,r1                  # Set up r11 with back chain 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.
                                       # Save CR here if necessary ...
        bl   _save64gpr_ctr_14_g # Save 64-bit nonvolatile gprs and fetch GOT ptr
       mflr    r31                     # Place GOT ptr in r31
                                       # 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 back chain word address
       b       _rest32gpr_26_x         # Restore 32-bit gprs and return
```

Figure 2-34. Prologue and Epilogue Sample Code

2.7.4 Profiling

This section shows a way of providing profiling (entry counting). An ABI-conforming system is not required to provide profiling; however if it does, this is one possible (not required) implementation.

If a function is to be profiled, it saves the link register in the LR save word of its caller's stack frame, loads into r0 a pointer to a word-aligned, one-word, static data area initialized to zero in which the _mcount routine is to maintain a count of the number of entries, and calls _mcount. For example, the code in Figure 2-34 can be inserted at the beginning of a function, before any other prologue code. The _mcount routine is required to restore the link register from the stack so that the profiling code (shown in Figure 2-35) can be inserted transparently, whether or not the profiled function saves the link register itself.

```
.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
```

Figure 2-35. Code for Profiling

NOTE

The value of the assembler expression symbol@l is the low-order 16 bits of the value of the symbol. The value of the expression symbol@ha is the high-order 16 bits of the value of the symbol, adjusted so that when it is shifted left by 16 bits and symbol@l is added to it, the resulting value is the value of the symbol. That is, symbol@ha compensates as necessary for the carry that may take place because of symbol@l being a signed quantity.

2.7.5 Data Objects

This section describes only objects with static storage duration. It excludes stack-resident objects because programs always compute their virtual addresses relative to the stack or frame pointers.

In the PowerPC Architecture, only load and store instructions access memory. Because PowerPC instructions cannot hold 32-bit addresses directly, a program normally computes an address into a register and accesses memory through the register. Symbolic references in absolute code put the symbols' values-or absolute virtual addresses-into instructions.

Position-independent instructions cannot contain absolute addresses. Instead, instructions that reference symbols hold the symbols' (signed) offsets into the global offset table. Combining the offset with the global offset table address in a general register (for example, r31 loaded in the sample prologue in Figure 2-33) gives the absolute address of the table entry holding the desired address.

Figure 2-36 through Figure 2-38 show sample assembly language equivalents to C language code for absolute and position-independent compilations. It is assumed that all shared objects are compiled position independent and only executable modules may be absolute. The code in the figures contains many redundant operations; it is intended to show how each C statement would have been compiled independently of its context.

C code	Assembly code
extern int src; extern int dst; extern int *ptr;	.extern src .extern dst .extern ptr

dst = src;	addis r6,r0,src@ha
	lwz r0,src@l(r6)
	addis r7,r0,dst@ha
	stw r0,dst@l(r7)

ptr = &dst;	addis r6,r0,dst@ha
	addi r0,r6,dst@l
	addis r7,r0,ptr@ha
	stw r0,ptr@l(r7)

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

Figure 2-36. Absolute Load and Store

NOTE

In the examples that follow, the assembly syntax symbol@got refers to the offset in the global offset table at which the value of symbol (that is, the address of the variable whose name is symbol) is stored, assuming that the offset is no larger than 16 bits. The syntax symbol@got@ha, symbol@got@h, and symbol@got@l refer to the high-adjusted, high, and low parts of that offset, when the offset may be greater than 16 bits.

Figure 2-37 shows small model position-independent load and store operations.

C code	Assembly code
extern int src; extern int dst; extern int *ptr;	.extern src .extern dst .extern ptr .text # Assumes GOT pointer in r31

dst = src;	lwz r6,src@got(r31)
	lwz r7,dst@got(r31)
	lwz r0,0(r6)
	stw r0,0(r7)

ptr = &dst;	lwz r0,dst@got(r31)
	lwz r7,ptr@got(r31)
	stw r0,0(r7)

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

Figure 2-37. Small Model Position-Independent Load and Store

Figure 2-38 shows large model position-independent load and store operations.

```
C code                                  Assembly code

extern int src;                         .extern src
extern int dst;                         .extern dst
int *ptr;                               .extern ptr

                                        .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)
```

Figure 2-38. Large Model Position-Independent Load and Store

2.7.6 Function Calls

Programs use the PowerPC **bl** instruction to make direct function calls. A bl instruction has a self-relative branch displacement that can reach 32 Mbytes in either direction. Hence, the use of a **bl** instruction to effect a call within an executable or shared object file limits the size of the executable or shared object file text segment.

A compiler normally generates the **bl** instruction to call a function as shown in Figure 2-39. The called function may be in the same module (executable or shared object) as the caller, or it may be in a different module. In the former case, the link editor resolves the symbol and the **bl** branches directly to the called function. In the latter case, the link editor cannot directly resolve the symbol. Instead, it treats the **bl** as a branch to glue code that it generates, and the dynamic linker modifies the glue code to branch to the function itself. See Section 4.3.4, “Procedure Linkage Table,” for more details.

Figure 2-39 shows a direct function call.

```
C code                                  Assembly code

extern void func();                     .extern func
func();                                 bl func
```

Figure 2-39. Direct Function Call

For indirect function calls, a **bctrl** instruction is used as shown in Figure 2-40 through Figure 2-42.

Figure 2-40 shows an absolute indirect function call.

```
C code                                  Assembly code

extern void func();                     .extern func
extern void (*ptr) ();                  .extern ptr

                                        .text

ptr = func;                             addis   r6,     r0,             func@ha
                                        addi    r0,     r6,             func@l(r6)
                                        addis   r7,     r0,             ptr@ha
                                        stw     r0,     ptr@l(r7)

(*ptr)();                               addis   r6,     r0,             ptr@ha
                                        lwz     r0,     ptr@l(r6)
                                        mtctr   r0
                                        bctrl
```

Figure 2-40. Absolute Indirect Function Call

Figure 2-41 shows a small model position-independent indirect function call.

```
C code                                  Assembly code

extern void func();                     .extern func
extern void (*ptr) ();                  .extern ptr

                                        .text

                                        # Assumes GOT pointer in r31

ptr = func;                             lwz     r0,     func@got(r31)
                                        lwz     r12,    ptr@got(r31)
                                        stw     r0,     0(r12)

(*ptr) ();                              lwz     r12,    ptr@got(r31)
                                        lwz     r0,     0(r12)
                                        mtctr   r0
                                        bctrl
```

Figure 2-41. Small Model Position-Independent Indirect Function Call

Figure 2-42 shows a large model position-independent indirect function call.

```
C code                                  Assembly code

extern void func();                     .extern func
extern void (*ptr) ();                  .extern ptr

                                        .text

                                        # Assumes GOT pointer in r31

ptr=func;                               addis   r11,    r31,            func@got@ha
                                        lwz     r0,     func@got@l(r11)
                                        addis   r12,    r31,            ptr@got@ha
                                        lwz     r12,    ptr@got@l(r12)
                                        stw     r0,     0(r12)

(*ptr) ();                              addis   r12,    r31,            ptr@got@ha
                                        lwz     r12,    ptr@got@l(r12)
                                        lwz     r0,     0(r12)
                                        mtctr   r0
                                        bctrl
```

Figure 2-42. Large Model Position-Independent Indirect Function Call

2.7.7 Branching

Programs use branch instructions to control their execution flow. As defined by the architecture, branch instructions hold a self-relative value with a 64-Mbyte range, allowing a jump to locations up to 32 Mbytes away in either direction.

Figure 2-43 shows the model for branch instructions.

C code

```
label:
    ...
    goto label;
```

Assembly code

```
.L01:
     ...
     b .L01
```

Figure 2-43. Branch Instruction, All Models

C switch statements provide multiway selection. When the case labels of a switch statement satisfy grouping constraints, the compiler implements the selection with an address table. The following examples use several simplifying conventions to hide irrelevant details:

- The selection expression resides in r12.
- The case label constants begin at zero.
- The case labels, the default, and the address table use assembly names .Lcase*i*, .Ldefault, and .Ltab, respectively.

Figure 2-44 shows absolute switch code.

C code

```
switch(j)
{
    case 0:
    ...
    case 1:
    ...
    case 3:
    ...
    default:
    ...
}
```

Assembly code

```
     cmplwi   r12, 4
     bge      .Ldefault
     slwi     r12, 2
     addis    r12, r12, .Ltab@ha
     lwz      r0, .Ltab@l(r12)
     mtctr    r0
     bctr
     .rodata
.Ltab:
     .long    .Lcase0
     .long    .Lcase1
     .long    .Ldefault
     .long    .Lcase3
     .text
```

Figure 2-44. Absolute Switch Code

Figure 2-45 shows the model for position-independent switch code.

```
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
```

Figure 2-45. Position-Independent Switch Code, All Models

2.7.8 Dynamic Stack Space Allocation

Stack frames are allocated dynamically on the program stack, depending on program execution, but individual stack frames can have static sizes. Nonetheless, the architecture supports dynamic allocation for those languages that require it. The mechanism for allocating dynamic space is embedded completely within a function and does not affect the standard calling sequence. Thus languages that need dynamic stack frame sizes can call C functions, and vice versa.

Figure 2-45 shows the stack frame before and after dynamic stack allocation. The local variables area is used for storage of function data, such as local variables, whose sizes are known to the compiler. This area is allocated at function entry and does not change in size or position during the function's activation.

The parameter list area holds overflow arguments passed in calls to other functions. (See the OTHER label in the algorithm in Parameter Passing earlier in this chapter.) Its size is also known to the compiler and can be allocated along with the fixed frame area at function entry. However, the standard calling sequence requires that the parameter list area begin at a fixed offset (8) from the stack pointer, so this area must move when dynamic stack allocation occurs.

Data in the parameter list area are naturally addressed at constant offsets from the stack pointer. However, in the presence of dynamic stack allocation, the offsets from the stack pointer to the data in the local variables area are not constant. To provide addressability, a frame pointer is established to locate the local variables area consistently throughout the function's activation. Dynamic stack allocation is accomplished by opening the stack just above the parameter list area. The following steps show the process in detail:

1. Sometime after a new stack frame is acquired and before the first dynamic space allocation, a new register, the frame pointer, is set to the value of the stack pointer. The frame pointer is used for references to the function's local, non-static variables.
2. The amount of dynamic space to be allocated is rounded up to a multiple of 16 bytes, so that 16-byte stack alignment is maintained.

3. The stack pointer is decreased by the rounded byte count, and the address of the previous stack frame (the back chain) is stored at the word addressed by the new stack pointer. This shall be accomplished atomically by using stwu rS,-length(r1) if the length is less than 32768 bytes, or by using **stwux rS,r1,r**space, where **r**S is the contents of the back chain word and rspace contains the (negative) rounded number of bytes to be allocated, as shown in Figure 2-46.

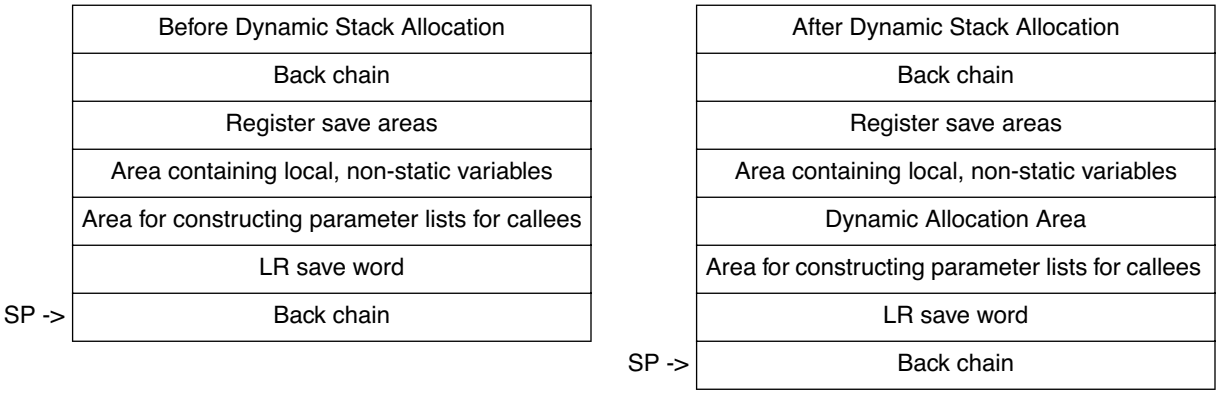

Figure 2-46. Dynamic Stack Space Allocation

The above process can be repeated as many times as desired within a single function activation. When it is time to return, the stack pointer is set to the value of the back chain, thereby removing all dynamically allocated stack space along with the rest of the stack frame. Naturally, a program must not reference the dynamically allocated stack area after it has been freed.

Even in the presence of signals, the above dynamic allocation scheme is safe. If a signal interrupts allocation, one of three things can happen:

- The signal handler can return. The process then resumes the dynamic allocation from the point of interruption.
- The signal handler can execute a non-local goto or a jump. This resets the process to a new context in a previous stack frame, automatically discarding the dynamic allocation.
- The process can terminate. Regardless of when the signal arrives during dynamic allocation, the result is a consistent (though possibly dead) process.

2.8 DWARF Definition

This section briefly describes some of the DWARF numbers and conventions used for e500 debugging.

2.8.1 DWARF Release Number

This section defines the debug with arbitrary record format (DWARF) debugging format for processors that implement the PowerPC architecture. The PowerPC ABI does not define a

debug format. However, all systems that do implement DWARF shall use the following definitions.

DWARF is a specification developed for symbolic, source-level debugging. The debugging information format does not favor the design of any compiler or debugger. For more information on DWARF, see the documents cited in the section Evolution of the ABI Specification in Chapter 1, "Introduction."

The DWARF definition requires some machine-specific definitions. The register number mapping needs to be specified for the PowerPC registers. In addition, the DWARF Version 2 specification requires processor-specific address class codes to be defined.

2.8.2 DWARF Register Number Mapping

Table 2-17 outlines the register number mapping for the registers and several of the SPRs and PMRs for the e500. Note that for all special purpose registers (SPRs), the DWARF register number is simply 100 plus the SPR number, as defined in the e500 documentation. For all performance monitor registers (PMRs), the DWARF register number is 2048 plus the PMR number. For kernel debuggers that need to display privileged registers, the DWARF register number for the MSR (the only non-SPR privileged register) is provided in Table 2-18.

Note that register numbers 0-31 refer to the 32 low-order bits of general purpose registers R0-R31, while register numbers 1200-1231 refer to the 32 high-order bits of R0-R31. These register numbers are used in both DWARF location expressions and in the DWARF call frame information.

For example, a compiler should emit the following for a call frame entry for a 64-bit save of register *N*:

DW_CFA_offset_extended 1200+*N*

DW_CFA_offset *N*

For DWARF attribute information, DW_OP_piece should be used to concatenate the two register values. For big-endian systems, the layout should be:

DW_OP_regx 1200+*N*

DW_OP_piece 4

DW_OP_reg*N*

DW_OP_piece 4

For little-endian systems, the layout should be:

DW_OP_regN

DW_OP_piece 4

DW_OP_regx 1200+N

DW_OP_piece 4

:These constructs are only needed for DWARF expressions for the entire 64-bit quantity.

Table 2-17. e500 Register Number Mapping

Register Name	Number	Abbreviation
Least-significant 32 bits of general-purpose registers 0–31	0–31	R0–R31
Most-significant 32 bits of general-purpose registers 0-31	1200-1231	R0-R31
Condition register	64	CR
Accumulator	99	ACC
Integer exception register	101	XER or SPR1
Link register	108	LR or SPR8
Count register	109	CTR or SPR9
Signal processing and embedded floating-point status and control register	612	SPEFSCR or SPR512
Embedded floating-point status and control register		EFSCR or SPR512

Table 2-18. e500 Privileged Register Number Mapping

Register Name	Number	Abbreviation
Machine state register	66	MSR
<any privileged SPR>	100+SPR#	—

The following table summarizes all of the DWARF register encodings for the PowerPC architecture.

Table 2-19. Summary of PowerPC Register Numbers

Register Name	Number	Abbreviation
Least-significant 32 bits of general-purpose registers	0–31	R0–R31
Floating-point registers (Not implemented on the e500)	32–63	F0–F31
Condition register	64	CR
Floating-point status and control register (Not implemented on the e500)	65	FPSCR
Machine state register	66	MSR
Accumulator	99	ACC
SPRs	100–1123	LR, CTR, etc.
AltiVec registers (Not implemented on the e500)	1124–1155	V0–V31
Reserved	1156–1199	
Most-significant 32 bits of general-purpose registers 0-31	1200-1231	R0-R31
Reserved	1232-2047	
Device control registers (Not implemented on the e500)	3072–4095	DCRs
Performance monitor registers	4096-5120	PMRs

2.8.3 Address Class Codes

The e500 processor family defines the address class codes described in Table 2-20.

Table 2-20. Address Class Code

Code	Value	Meaning
ADDR_none	0	No class specified

2.9 SPE Register Core Dump Image Specification

SPE registers are dumped into a core file note section like other register groupings. The details for the note section are shown in Table 2-21.

Table 2-21. SPE Register State Note Section Information

NAMESZ = 4
DATASZ = 268
TYPE = 21 /* NT_SPEREGSET */
NAME = “CORE”
DATA = an instance of a “struct speregset”

The data structure is defined as:

```
struct speregset {
        uint64_t GPR[32];       // We dump the full 64-bit registers
        uint64_t acc;           // We dump the accumulator
        uint32_t spefscr;       // We dump SPR512
}
```

Chapter 3
Object Files

3.1 ELF Header

3.1.1 Machine Information

For file identification in e_ident, processors that implement the PowerPC architecture require the values shown in Table 3-1.

Table 3-1. PowerPC Identification, e_ident []

Position	Value	Comments
e_ident[EI_CLASS]	ELFCLASS32	For all 32-bit implementations
e_ident[EI_DATA]	ELFDATA2MSB	For all Big-Endian implementations
e_ident[EI_DATA]	ELFDATA2LSB	For all Little-Endian implementations

The ELF header's e_flags member holds bit flags associated with the file. Because processors that implement the PowerPC architecture define no flags, this member contains zero.

The name EF_PPC_EMB and the value 0x8000_0000 are reserved for use in embedded systems.

Processor identification resides in the ELF header's e_machine member and must have the value 20, defined as the name EM_PPC.

3.2 Sections

3.2.1 Special Sections

Various sections hold program and control information. The sections listed in Table 3-2 are used by the system and have the types and attributes shown.

NOTE

The .plt section in PowerPC object files is of type SHT_NOBITS, not SHT_PROGBITS as on most other processors' binaries.

NOTE

The SHT_ORDERED section type specifies that the link editor is to sort the entries in this section based on the sum of the symbol and addend values specified by the associated relocation entries. Entries without associated relocation entries shall be appended to the end of the section in an unspecified order. SHT_ORDERED is defined as SHT_HIPROC, the first value reserved in the System V ABI for processor-specific semantics.

NOTE

The section names for .sdata2 and .sbss2 have been changed to .PPC.EMB.sdata2 and .PPC.EMB.sbss2, to comply with the latest System V ABI.

Table 3-2. Special Section Types and Attributes

Name	Type	Attributes
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.plt	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_EXECINSTR
.sdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.PPC.EMB.sdata2	SHT_PROGBITS	SHF_ALLOC and possibly SHF_WRITE (see Table 3-4)
.PPC.EMB.sdata0	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.sbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.PPC.EMB.sbss2	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.PPC.EMB.sbss0	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.PPC.EMB.apuinfo	SHT_NOTE	0
.PPC.EMB.seginfo	SHT_PROGBITS	0

Special sections are described in Table 3-4.

Table 3-3. Special Section Descriptions

Name	Description
.got	Holds the Global Offset Table, or GOT. See Section 2.7, “Coding Examples,” and Section 4.3.2, “Global Offset Table,” for more information.
.plt	Holds the procedure linkage table. See Section 4.3.4, “Procedure Linkage Table.”
.sdata	Holds initialized small data that contribute to the program memory image. See Section 3.3.1, “Small Data Area (.sdata and .sbss),” for details.
.PPC.EMB.sdata2	Intended to hold initialized read-only small data that contribute to the program memory image. The section can, however, be used to hold writable data. If a linker 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 3.3.1, “Small Data Area (.sdata and .sbss),” for more details.
.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. Section 3.3.3, “Small Data Area 0 (.PPC.EMB.sdata0 and .PPC.EMB.sbss0),” for more details.
.sbss	Holds uninitialized small data that contribute to the program memory image. The system sets the data to zeros when the program begins to run. Section 3.3.1, “Small Data Area (.sdata and .sbss).”
.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 3.3.2, “Small Data Area 2 (.PPC.EMB.sdata2 and .PPC.EMB.sbss2),” for details.
.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 3.3.3, “Small Data Area 0 (.PPC.EMB.sdata0 and .PPC.EMB.sbss0),” for more details.
.PPC.EMB.apuinfo	Contains records describing which APUs are required for this program to execute properly. See Section 3.6, “APU Information Section.” .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 3.7, “ROM Copy Segment Information Section,” for more details.

NOTE

This ABI shares most of the linkage conventions of the classic PowerPC ABI and/or EABI, including sdata2/sbss2 support (although these sections have been prefixed with .PPC.EMB in order to comply with the latest System V ABI specifications).

3.3 Small Data Areas

Three distinct small data areas, each possibly containing both initialized and zero-initialized data, are supported by this ABI, and are summarized in Table 3-4.

Table 3-4. Small Data Areas Summary

Section Names	Register/Value Used	Symbol	Can It be Used for Addressing in Shared Objects?
.sdata, .sbss	r13	_SDA_BASE_	yes, only for local data
.PPC.EMB.sdata2, .PPC.EMB.sbss2	r2	_SDA2_BASE_	no
.PPC.EMB.sdata0, .PPC.EMB.sbss0	0	n.a.	no

All three areas can contain at most 64 Kbytes of data items. All areas may hold both local and global data items in executable files. In shared object files, .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," single-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 Kbytes of data items that can be addressed in a single instruction: two 64-Kbyte regions that can be placed anywhere in the address space (but typically in standard locations—see Section 3.3.1, "Small Data Area (.sdata and .sbss)"), and one 64-Kbyte region straddling address 0 (32 Kbytes at addresses 0xFFFF_8000 through 0xFFFF_FFFF, and 32 Kbytes at addresses 0x0000_0000–0x0000_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 or shared object, and some of the code that makes up the file must be recompiled with a smaller value for the size criterion.

Note that 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.

3.3.1 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.

In both shared object and executable files, the small data area straddles the boundary between initialized and uninitialized data in the data segment of the file. The usual order of sections in the data segment, some of which may be empty, is:

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

Only data items with local (non-global) 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 Kbytes, and less if the global offset table is large.

For executable files, up to 64 Kbytes 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 executable files, 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.

3.3.2 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 3.2.1, "Special Sections").

The sum of the sizes of sections .PPC.EMB.sdata2 and .PPC.EMB.sbss2 in an object file shall not exceed 64 Kbytes. 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 file.

If an executable file contains a .PPC.EMB.sdata2 section or a .PPC.EMB.sbss2 section, then a linker 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 linker shall set _SDA2_BASE_ to 0.

3.3.3 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), as all addressing can be relative to address 0 (an address register encoding of r0 means the value 0 in PowerPC load and store instructions).

The sum of the sizes of sections .PPC.EMB.sdata0 and .PPC.EMB.sbss0 in an object file shall not exceed 64 Kbytes. 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 file.

3.4 Tags

The e500 ABI has removed any requirements for tag support, as the functionality of tags can now be provided via DWARF debugging information.

3.5 Symbol Table

3.5.1 Symbol Values

If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for the file will contain an entry for that symbol. The st_shndx member of that symbol table entry contains SHN_UNDEF. This informs the dynamic linker that the symbol definition for that function is not contained in the executable file itself. If that symbol has been allocated a procedure linkage table entry in the executable file, and the st_value member for that symbol table entry is nonzero, the value is the virtual address of the first instruction of that procedure linkage table entry. Otherwise, the st_value member contains zero. This procedure linkage table entry address is used by the dynamic linker in resolving references to the address of the function. See Section 4.3.3, “Function Addresses,” for details.

3.6 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 Figure 3-1.

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

Figure 3-1. Typical Elf Note Section Format

For the .PPC.EMB.apuinfo section, the name shall be “APUinfo\0”, the type shall be 2 (as type 1 is already reserved for ELF_NOTE_ABI), and the data shall contain a series of words containing APU information, one per word. 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.

Example:

```
Object file a.o:

0   | 0x00000008 | # 8 bytes in “APUinfo\0”
4   | 0x0000000C | # 12 bytes (3 words) of APU information
8   | 0x00000002 | # NOTE type 2
12  | 0x41505569 | # ASCII for “APUi”
16  | 0x6e666f00 | # ASCII for “nfo\0”
20  | 0x00010001 | # APU #1, revision 1
24  | 0x00020003 | # APU #2, revision 3
28  | 0x00040001 | # APU #4, revision 1

Object file b.o:

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
16  | 0x00010002 | # APU #1, revision 2
20  | 0x00040001 | # APU #4, revision 1
```

Linkers should merge all .PPC.EMB.apuinfo sections in the individual object files into one, with merging of per-APU information:

```
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
16  | 0x00010002 | # APU #1, revision 2
20  | 0x00020003 | # APU #2, revision 3
24  | 0x00040001 | # APU #4, revision 1
```

Note that it is assumed that a later revision of any APU is compatible with an earlier one, but not vice-versa. 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.

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

The APU identifiers as of April 2003 are listed in Table 3-5. Please see “Motorola Book E Implementation Standards: APU ID Reference” for up-to-date assignments.

Table 3-5. APU Identifiers as of April, 2003

APU identifier (16 bits)	APU
0x0-0x3e	Reserved
0x3f	Motorola AltiVec APU [1]
0x40	Motorola Book E isel APU
0x41	Motorola Book E Performance Monitor APU
0x42	Motorola Book E Machine-check APU
0x43	Motorola Book E Cache-locking APU
0x44-0xff	Reserved
0x100	e500 SPE APU
0x101	e500 SPFP PU
0x102	e500 Branch-locking APU
0x103-0xffff	Reserved

[1] Although AltiVec predates the concept of Book E APUs, AltiVec can be considered an APU.

3.7 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 static linker 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.

In the section header for .PPC.EMB.seginfo:

- sh_link shall be either SHN_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_entsize shall be 12.
- sh_flags, sh_addr, sh_info, and sh_addralign shall all be 0.

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:

- sg_indx shall be 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 shall be a bit mask of flags. The only allowed flag shall be as shown in Table 3-6.

Table 3-6. Allowed Flag

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

- sg_name shall be 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 shall contain 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.
- EXTENDED 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 static linker 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.

3.8 Relocation

3.8.1 Relocation Types

Note that support for dynamic linking, the GOT, and the PLT are considered EXTENDED conformance.

Relocation entries describe how to alter the instruction and data relocation fields shown in Figure 3-2 (bit numbers appear in the lower box corners; Little-Endian byte numbers appear in the upper right box corners; Big-Endian numbers appear in the upper left box corners).

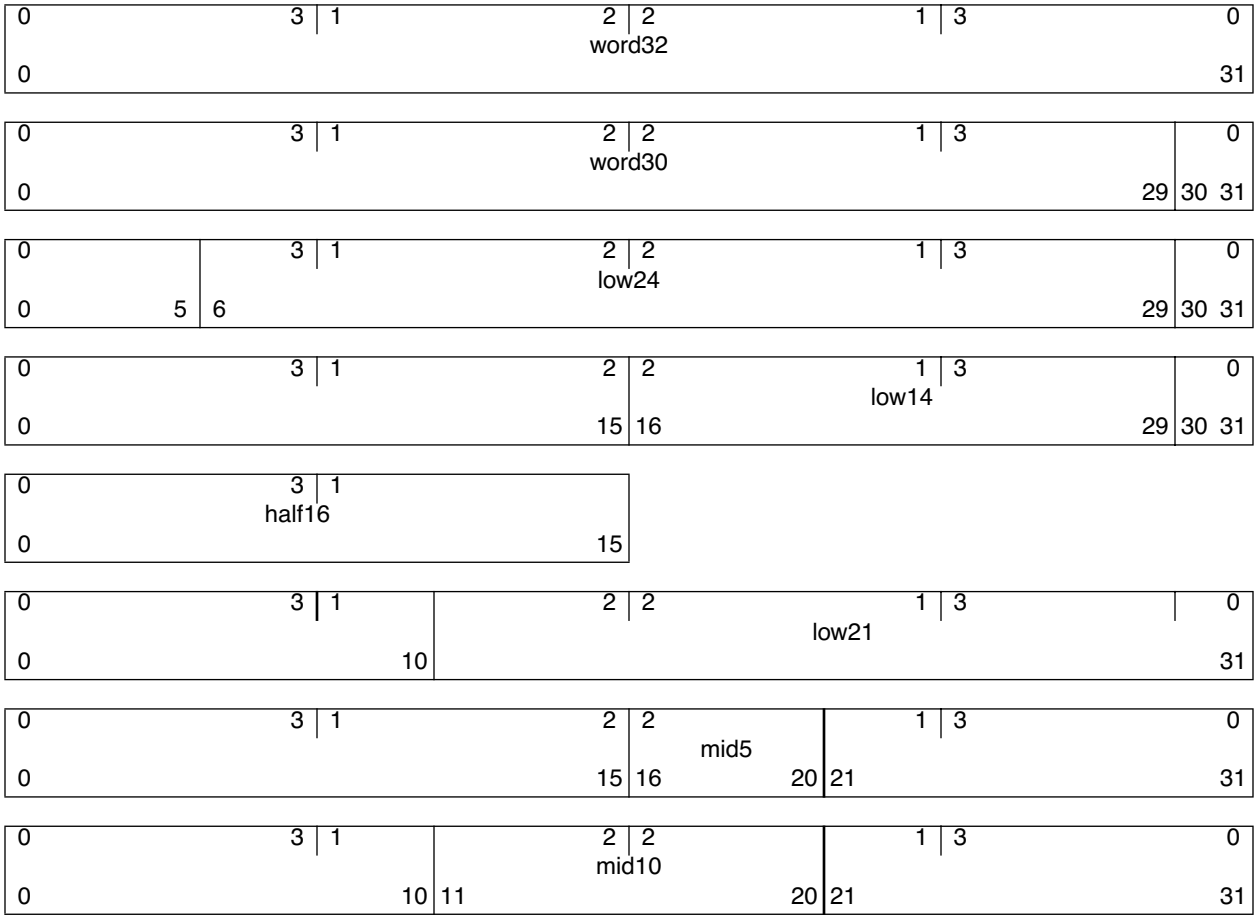

Figure 3-2. Relocation Fields

Table 3-7 describes relocation fields.

Table 3-7. Relocation Field Descriptions

Field	Descriptions
word32	Specifies a 32-bit field occupying 4 bytes, the alignment of which is 4 bytes unless otherwise specified.
word30	Specifies a 30-bit field contained within bits 0–29 of a word with 4-byte alignment. The two least significant bits of the word are unchanged.
low24	Specifies a 24-bit field contained within a word with 4-byte alignment. The six most significant and the two least significant bits of the word are ignored and unchanged (for example, “Branch” instruction).
low14	Specifies a 14-bit field contained within a word with 4-byte alignment, comprising a conditional branch instruction. The 14-bit relative displacement in bits 16–29, and possibly the “branch prediction bit” (bit 10), are altered; all other bits remain unchanged.
half16	Specifies a 16-bit field occupying 2 bytes with 2-byte alignment (for example, the immediate field of an Add Immediate instruction).
low21	Specifies a 21-bit field occupying the least significant bits of a word with 4-byte alignment. Note that the classic EABI had a different definition for this relocation type.
mid5	Specifies a 5-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.
mid10	Specifies a 10-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.

NOTE

The classic EABI stated that several relocation entry types (uword32, ulow21 (here called low21), and uhalf16) did not have any alignment restrictions. This ABI requires alignment for these relocations unless otherwise noted in the relocation description.

Calculations in Table 3-9 assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link editor merges one or more relocatable files to form the output. It first determines how to combine and locate the input files, next it updates the symbol values, and then it performs relocations. Relocations applied to executable or shared object files are similar and accomplish the same result. The notations used in Table 3-9 are described in Table 3-8.

Table 3-8. Notation Conventions

Notation	Description
A	Represents the addend used to compute the value of the relocatable field.
B	Represents the base address at which a shared object 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 at which the address of the relocation entry's symbol will reside during execution. See Section 2.7, "Coding Examples," and Section 4.3.2, "Global Offset Table."
L	Represents the section offset or address of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See Section 4.3.4, "Procedure Linkage Table," for more information. P Represents the place (section offset or address) of the storage unit being relocated (computed using r_offset).
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.
T	Represents the offset from _SDA_BASE_ to where in .sdata the linker placed the address of the symbol whose index is in r_info. See R_PPC_EMB_SDA_I16 description below.
U	Represents the offset from _SDA2_BASE_ to where in .PPC.EMB.sdata2 the linker placed the address of the symbol whose index is in r_info. See R_PPC_EMB_SDA2_I16 description below.
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 linker placed the symbol whose index is in r_info. This is a generalized notation for the T and U cases.
Y	Represents a 5-bit value for the base register for the section where the linker 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.

Relocation entries apply to halfwords or words. In either case, the r_offset value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. Processors that implement the PowerPC architecture use only the Elf32_Rela relocation entries with explicit addends. For relocation entries, the r_addend member serves as the relocation addend. In all cases, the offset, addend, and the computed result use the byte order specified in the ELF header.

The following general rules apply to the interpretation of the relocation types in Table 3-9:

- + and - denote 32-bit modulus addition and subtraction, respectively.
  || denotes concatenation of bits or bitfields.

  >> denotes arithmetic right-shifting (shifting with sign copying) of the value of the left operand by the number of bits given by the right operand.

- 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.
- #hi(value) and #lo(value) denote the most and least significant 16 bits, respectively, of the indicated value. That is, #lo(x)=(x & 0xFFFF) and #hi(x)=((x>> 16) & 0xFFFF). The “high adjusted” value, #ha (value), compensates for #lo() being treated as a signed number: #ha(x) = (((x >> 16) + ((x & 0x8000) ? 1 : 0)) & 0xFFFF).
- Reference in a calculation to the value G implicitly creates a GOT entry for the indicated symbol.
- _SDA_BASE_ is 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. Similarly, _SDA2_BASE_ is a symbol defined by the link editor whose value in executable programs is an address within the small data 2 area. See Section 3.3, “Small Data Areas,” for more details.
- Optional relocation types (those provided for dynamic linking and for compatibility with existing vendor-defined relocations) are marked with a + after their name.
- 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.

Note that the relocation types in Table 3-9 contain relocations from the classic ABI as well as the classic EABI, but some have been renamed slightly.

Table 3-9. Relocation Types

Name	Value	Field	Calculation
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

Table 3-9. Relocation Types (continued)

Name	Value	Field	Calculation
R_PPC_REL14_BRNTAKEN	13	low14*	(S + A - P) >> 2
R_PPC_GOT16+	14	half16*	G + A
R_PPC_GOT16_LO+	15	half16	#lo(G + A)
R_PPC_GOT16_HI+	16	half16	#hi(G + A)
R_PPC_GOT16_HA+	17	half16	#ha(G + A)
R_PPC_PLTREL24+	18	low24*	(L + A - P) >> 2
R_PPC_COPY+	19	none	none
R_PPC_GLOB_DAT+	20	word32	S + A
R_PPC_JMP_SLOT+	21	none	See below
R_PPC_RELATIVE	22	word32	B + A
R_PPC_LOCAL24PC+	23	low24*	see below
R_PPC_UADDR32	24	word32	S + A
R_PPC_UADDR16	25	half16*	S + A
R_PPC_REL32	26	word32	S + A - P
R_PPC_PLT32+	27	word32	L + A
R_PPC_PLTREL32+	28	word32	L + A - P
R_PPC_PLT16_LO+	29	half16	#lo(L + A)
R_PPC_PLT16_HI+	30	half16	#hi(L + A)
R_PPC_PLT16_HA+	31	half16	#ha(L + A)
R_PPC_SDAREL16	32	half16*	S + A - _SDA_BASE_
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
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_SDA_I16	106	half16*	T
R_PPC_EMB_SDA2_I16	107	half16*	U
R_PPC_EMB_SDA2REL	108	half16*	S + A - _SDA2_BASE_
R_PPC_EMB_SDA21	109	low21	Y ll (X + A). See below

Table 3-9. Relocation Types (continued)

Name	Value	Field	Calculation
R_PPC_EMB_MRKREF	110	none	See below
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 below
R_PPC_EMB_RELSDA	116	half16*	X + A. See below
R_PPC_EMB_RELOC_120+	120	half16*	S + A
R_PPC_EMB_RELOC_121+	121	half16*	Same calculation as U, except that the value 0 is used instead of _SDA2_BASE_.
R_PPC_DIAB_SDA21_LO+	180	half21	Y ll #lo(X + A)
R_PPC_DIAB_SDA21_HI+	181	half21	Y ll #hi(X + A)
R_PPC_DIAB_SDA21_HA+	182	half21	Y ll #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)
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 ll ((#lo(X + A)) >> 3)
R_PPC_EMB_SPE_WORD_SDA	214	mid10*	Y ll ((#lo(X + A)) >> 2)
R_PPC_EMB_SPE_HALF_SDA	215	mid10*	Y ll ((#lo(X + A)) >> 1)

Relocation values not in Table 3-9 and values outside of the range 101–200 are reserved. Values in the range 101–200 and names beginning with R_PPC_EMB_ are assigned for embedded system use. Values in the range 201–300 are assigned for use by APUs.

The relocation types in which the names include _BRTAKEN or _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.

Relocation types with special semantics are described in Table 3-10.

Table 3-10. Relocation Types with Special Semantics

Name	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	Refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table. There is an implicit assumption that the procedure linkage table for a module will be within +/- 32 Mbytes of an instruction that branches to it, so that the R_PPC_PLTREL24 relocation type is the only one needed for relocating branches to procedure linkage table entries.
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 object file and in a shared object. 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	Resembles R_PPC_ADDR32, except that it sets a global offset table entry to the address of the specified symbol. This special relocation type allows one to determine 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 Figure 4-3 "Procedure Linkage Table Example").
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	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_PPC_ADDR* types, except that the datum to be relocated is allowed to be unaligned.
R_PPC_EMB_SDA_I16	Instructs a linker 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 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.

Table 3-10. Relocation Types with Special Semantics (continued)

Name	Description
R_PPC_EMB_SDA2_I16	This instructs a linker 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.
R_PPC_EMB_SDA21	The most significant 11 bits at the address pointed to by the relocation entry shall be unchanged. If the symbol whose index is in r_info is contained in .sdata or .sbss, then a linker shall place in the next most significant 5 bits the value 13 (for GPR13); if the symbol is in .PPC.EMB.sdata2 or .PPC.EMB.sbss2, then the linker shall place in those 5 bits the value 2 (for GPR2); if the symbol is in .PPC.EMB.sdata0 or .PPC.EMB.sbss0, then the linker shall place in those 5 bits the value 0 (for GPR0); 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.
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, a linker shall not apply a relocation action to a location because of this type. This relocation type is used to prevent a linker that does section garbage collecting from deleting an important but otherwise unreferenced section.
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. A linker shall replace bits starting at the bit position for the specified length with the value of the symbol, treated as a signed entity.
R_PPC_EMB_RELSDA	The linker 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.

Chapter 4
Program Loading and Dynamic Linking

4.1 Program Loading—Extended Conformance

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When-and if-the system physically reads the file depends on the program's execution behavior, system load, and so on. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore, delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose offsets and virtual addresses are congruent, modulo the page size. Virtual addresses and file offsets segments are congruent modulo 64 Kbytes (0x1_0000) or larger powers of 2. Although 4096 bytes is currently the page size, this allows files to be suitable for paging even if implementations appear with larger page sizes. The value of the p_align member of each program header in a shared object file must be 0x1_0000. Figure 4-1 is an example of an executable file assuming an executable program linked with a base address of 0x1000_0000 (32 Mbytes).

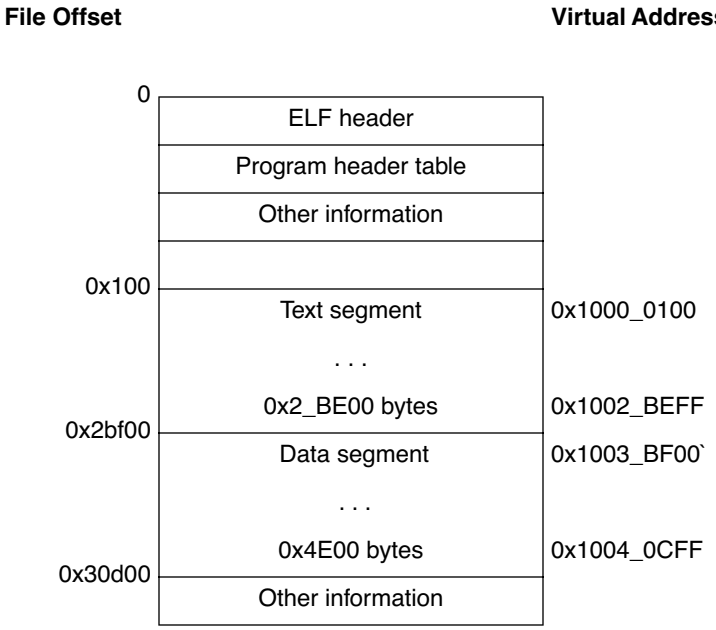

Figure 4-1. Executable File Example

Table 4-1. Program Header Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2_BF00
p_vaddr	0x1000_0100	0x1003_BF00
p_paddr	Unspecified	Unspecified
p_filesz	0x2_BE00	0x4E00
p_memsz	0x2_BE00	0x5E24
p_flags	PF_R+PF_X	PF_R+PF_W
p_align	0x1_0000	0x1_0000

Although the file offsets and virtual addresses are congruent modulo 64 Kbytes for both text and data, up to four file pages can hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.
- The last text page may hold a copy of the beginning of data.
- The first data page may have a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces memory permissions as if each segment were complete and separate; segment addresses are adjusted to ensure that each logical page in the address space has a single set of permissions. In the example in Figure 4-1, the file region holding the end of text and the beginning of data is mapped twice; at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if the last data page of a file includes information not in the logical memory page, the extraneous data must be set to zero, rather than to the unknown contents of the executable file. Impurities in the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for the program in Figure 4-1 is presented in Figure 4-2, assuming 4096 (0x1000) byte pages.

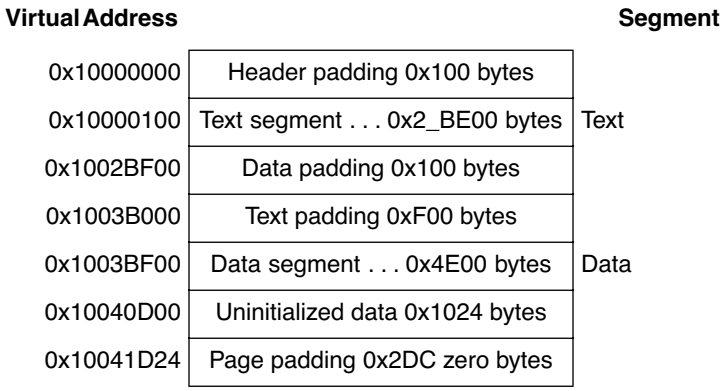

Figure 4-2. Process Image Segments

One aspect of segment loading differs between executable files and shared objects. Executable file segments may contain absolute code. For the process to execute correctly, the segments must reside at the virtual addresses assigned when building the executable file, with the system using the p_vaddr values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This allows a segment's virtual address to change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the relative positions of the segments. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. Table 4-2 shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also shows the base address computations.

Table 4-2. Shared Object Segment Example

Source	Text	Data	Base Address
File	0x00_0200	0x02_A400	
Process 1	0x10_0200	0x12_A400	0x10_0000
Process 2	0x20_0200	0x22_A400	0x20_0000
Process 3	0x30_0200	0x32_A400	0x30_0000
Process 4	0x40_0200	0x42_A400	0x40_0000

4.2 Program Interpreter—Extended Conformance

A program shall not specify a program interpreter other than /usr/lib/ld.so.1.

4.3 Dynamic Linking—Extended Conformance

4.3.1 Dynamic Section

Dynamic section entries give information to the dynamic linker. Some of this information is processor-specific, including the interpretation of some entries in the dynamic structure.

Table 4-3. Dynamic Section Entry Descriptions

Entry	Description
DT_PLTGOT	This entry's d_ptr member gives the address of the first byte in the procedure linkage table (.PLT in Figure 4-3).
DT_JMPREL	As explained in the System V ABI, this entry is associated with a table of relocation entries for the procedure linkage table. For processors implementing the PowerPC architecture, this entry is mandatory both for executable and shared object files. Moreover, the relocation table's entries must have a one- to-one correspondence with the procedure linkage table. The table of DT_JMPREL relocation entries is wholly contained within the DT_RELA referenced table. See Section 4.3.4, "Procedure Linkage Table," for more information.

4.3.2 Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

When the dynamic linker creates memory segments for a loadable object file, it processes the relocation entries, some of which will be of type R_PPC_GLOB_DAT, referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the global offset table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

A global offset table entry provides direct access to the absolute address of a symbol without compromising position-independence and sharability. Because the executable file and shared objects have separate global offset tables, a symbol may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution. The dynamic linker may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses. A global offset table's format and interpretation

are processor specific. For processors implementing the PowerPC architecture, the symbol _GLOBAL_OFFSET_TABLE_ may be used to access the table. The symbol may reside in the middle of the .got section, allowing both positive and negative "subscripts" into the array of addresses. Four words in the global offset table are reserved:

- The word at _GLOBAL_OFFSET_TABLE_[-1] shall contain a **blrl** instruction (see the text relating to Figure 2-34 "Prologue and Epilogue Sample Code").
- The word at _GLOBAL_OFFSET_TABLE_[0] is set by the link editor to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image.
- The word at _GLOBAL_OFFSET_TABLE_[1] is reserved for future use.
- The word at _GLOBAL_OFFSET_TABLE_[2] is reserved for future use.

The global offset table resides in the ELF .got section.

4.3.3 Function Addresses

References to the address of a function from an executable file and the shared objects associated with it need to resolve to the same value. References from within shared objects will normally be resolved by the dynamic linker to the virtual address of the function itself. References from within the executable file to a function defined in a shared object will normally be resolved by the link editor to the address of the procedure linkage table entry for that function within the executable file.

To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry. See Section 3.5.1, "Symbol Values," for details. The dynamic linker treats such symbol table entries specially. If the dynamic linker is searching for a symbol and encounters a symbol table entry for that symbol in the executable file, it normally follows these rules:

- If the st_shndx member of the symbol table entry is not SHN_UNDEF, the dynamic linker has found a definition for the symbol and uses its st_value member as the symbol's address.
- If the st_shndx member is SHN_UNDEF and the symbol is of type STT_FUNC and the st_value member is not zero, the dynamic linker recognizes this entry as special and uses the st_value member as the symbol's address.
- Otherwise, the dynamic linker considers the symbol to be undefined within the executable file and continues processing.

Some relocations are associated with procedure linkage table entries. These entries are used for direct function calls rather than for references to function addresses. These relocations

are not treated in the special way described above because the dynamic linker must not redirect procedure linkage table entries to point to themselves.

4.3.4 Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. The dynamic linker determines the destinations' absolute addresses and modifies the procedure linkage table's memory image accordingly. The dynamic linker can thus redirect the entries without compromising the position- independence and sharability of the program's text. Executable files and shared object files have separate procedure linkage tables.

For processors implementing the PowerPC architecture, the procedure linkage table (the .plt section) is not initialized in the executable or shared object file. Instead, the link editor simply reserves space for it, and the dynamic linker initializes it and manages it according to its own, possibly implementation- dependent needs, subject to the following constraints:

- The first 18 words (72 bytes) of the procedure linkage table are reserved for use by the dynamic linker. There shall be no branches from the executable or shared object into these first 18 words.
- If the executable or shared object requires N procedure linkage table entries, the link editor shall reserve 3*N words (12*N bytes) following the 18 reserved words. The first 2*N of these words are the procedure linkage table entries themselves. The static linker directs calls to bytes (72 + (i-1)*8), for i between 1 and N inclusive. The remaining N words (4*N bytes) are reserved for use by the dynamic linker.

As mentioned before, a relocation table is associated with the procedure linkage table. The DT_JMPREL entry in the _DYNAMIC array gives the location of the first relocation entry. The relocation table's entries parallel the procedure linkage table entries in a one-to-one correspondence. That is, relocation table entry 1 applies to procedure linkage table entry 1, and so on. The relocation type for each entry shall be R_PPC_JMP_SLOT, the relocation offset shall specify the address of the first byte of the associated procedure linkage table entry, and the symbol table index shall reference the appropriate symbol.

To illustrate procedure linkage tables, Figure 4-3 shows how the dynamic linker might initialize the procedure linkage table when loading the executable or shared object.

```
.PLT:
```

```
.PLTresolve:
    addis       r12,r0,dynamic_linker@ha
    addi        r12,r12,dynamic_linker@l
    mtctr       r12
    addis       r12,r0,symtab_addr@ha
    addi        r12,r12,symtab_addr@l
    bctr        .PLTcall:
    addis       r11,r11,.PLTtable@ha
    lwz         r11,.PLTtable@l(r11)
    mtctr       r11
    bctr
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
.PLT1:
    addi        r11,r0,4*0
    b           .PLTresolve
...

.PLTi:
    addi        r11,r0,4*(i-1)
    b           .PLTresolve
...
.PLTN:
    addi        r11,r0,4*(N-1)
    b           .PLTresolve
.PLTtable:
    <N word table begins here>
```

Figure 4-3. Procedure Linkage Table Example

Following the steps below, the dynamic linker and the program cooperate to resolve symbolic references through the procedure linkage table. Again, these steps are shown only for explanation. The precise execution-time behavior of the dynamic linker is not specified.

1. As shown above, all procedure linkage table entries initially transfer to .PLTresolve, allowing the dynamic linker to gain control at the first execution of each table entry. For example, assume the program calls name, which transfers control to the label .PLTi. The procedure linkage table entry loads into r11 four times the index of the relocation entry for .PLTi and branches to .PLTresolve, which then calls into the dynamic linker with a pointer to the symbol table for the object in r12.
2. The dynamic linker finds relocation entry i corresponding to the index in r11. It will have type R_PPC_JMP_SLOT, its offset will specify the address of .PLTi, and its symbol table index will reference name.
3. Knowing this, the dynamic linker finds the symbol's "real" value. It then modifies the code at .PLTi in one of two ways. If the target symbol is reachable from .PLTi by a branch instruction, it overwrites the "addi r11,r0,4*(i-1)" instruction at .PLTi with a branch to the target. On the other hand, if the target symbol is not reachable from .PLTi, the dynamic linker loads the target address into word .PLTtable+4*(i-1) and overwrites the "b .PLTresolve" with a "b .PLTcall".

4. Subsequent executions of the procedure linkage table entry will transfer control directly to the function, either directly or by using .PLTcall, without invoking the dynamic linker.

For PLT indexes greater than or equal to 2^13, only the even indexes shall be used and four words shall be allocated for each entry. If the above scheme is used, this allows four instructions for loading the index and branching to .PLTresolve or .PLTcall, instead of only two. The LD_BIND_NOW environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker resolves the function call binding at load time, before transferring control to the program. That is, the dynamic linker processes relocation entries of type R_PPC_JMP_SLOT during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

NOTE

Lazy binding generally improves overall application performance because unused symbols do not incur the dynamic linking overhead. Nevertheless, the following two situations make lazy binding undesirable for some applications:

- The initial reference to a shared object function takes longer than subsequent calls because the dynamic linker intercepts the call to resolve the symbol, and some applications cannot tolerate this unpredictability.
- If an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.

Chapter 5
Libraries

5.1 System Library (libsys)

As mentioned in the *System V Application Binary Interface*, Edition 4.1, the system library (libsys) has been deprecated. The routines (required or optional) that were provided by libsys are now provided by the C library (libc).

5.2 C Library (libc)

5.2.1 C Library Conformance with Generic ABI

The C Library should conform to the specifications of the *System V Application Binary Interface*, Edition 4.1. This also includes the discussion of Global Data Symbols (for example, `errno`, `optarg`, and `optind`) as well as Application Constraints (`environ` and `_lib_version`).

Note that the `malloc` routine must return a pointer that is at least 8-byte (double-word) aligned, as the returned pointer may be used for storing data items that require 8-byte alignment. If data items require alignment at a larger granularity than 8 bytes, a function like memalign should be called instead of malloc.

5.2.2 Processor-Specific Required Routines

There are four different classes of processor-specific required routines: the save and restore routines, used to reduce the size of function prologs and epilogs; the `_ _va_arg` routine, used for variable-argument handling; routines for handling 64-bit integral types (`long long`); and routines for querying which revisions of which APUs an implementation supports.

5.2.2.1 Save and Restore Routines

All of the save and restore routines discussed in Section 2.7.3, “Register Saving and Restoring Functions,” are required to be in libc. Each set of entry points should be provided by distinct object files within libc, to minimize the amount of code that is placed in the text section from this library. Note that these routines use unusual calling conventions due to

their special purpose. The routines in Figure 5-1 must be provided, where the suffix "*_m*" designates the first register to be saved. That is, to save registers 18–31 using 32-bit saves, one would call `save32gpr_18`.

```
save32gpr_m       save64gpr_m          rest32gpr_m          rest64gpr_m
save32gpr_m_g     save64gpr_m_g        rest32gpr_m_x        rest64gpr_ctr_m
                  save64gpr_ctr_m_g    rest32gpr_m_t        rest64gpr_m_x
                                                            rest64gpr_m_t
```

Figure 5-1. Required Save and Restore Routines

5.2.2.2 Variable-Argument Routine

An implementation must provide the following routine, shown in Figure 5-2, in libc to support variable-argument handling.

```
_ _va_arg
```

Figure 5-2. libc Required Variable-Argument Routines

void * _ _va_arg(va_list argp, _va_arg_type type)

This function is used by the va_arg macros of <stdarg.h> and <varargs.h>, and it returns a pointer to the next argument specified in the variable argument list argp. A variable argument list is an array of one structure, as shown below.

```
void * _ _va_arg(va_list argp, _va_arg_type type)

/* overflow_arg_area is initially the address at which the
* first arg passed on the stack, if any, was stored.
*
* reg_save_area is the start of where r3:r10 were stored.
* reg_save_area must be doubleword aligned.
*
* Since only 32-bit values are stored,
* the eight 32-bit values start at reg_save_area.
*
*/

typedef struct {
    char gpr; /* index into the array of 8 GPRs
         * stored in the register save area
         * gpr=0 corresponds to r3,
         * gpr=1 to r4, etc.
         */
    char fpr;/* UNUSED, but kept for compatibility */
    char *overflow_arg_area;
         /* location on stack that holds
         * the next overflow argument
         */
    char *reg_save_area;
         /* where r3:r10 are stored */
} va_list[1];
```

The argument is assumed to be of type `type`. The types are described in Table 5-1.

Table 5-1. Argument Types

Type	Description
0	arg_ARGPOINTER A struct, union, or long double argument represented in the PowerPC calling conventions as a pointer to (a copy of) the object.
1	arg_WORD A 32-bit aligned word argument, any of the simple integer types, a single-precision float type, or a pointer to an object of any type.
2	arg_DOUBLEWORD A long long argument, a `__ev64_opaque__` argument, or a double argument held in two 32-bit registers. This includes float arguments that are widened to double arguments.
3	arg_ARGREAL (reserved)

Note that float arguments are converted to double arguments (per the C language's rules on argument widening). All __ev64_opaque__ variables are passed in two registers, in the same manner as long long type variables, for variable-argument functions. Thus, floats, doubles, `__ev64_opaque` , and long long type variables are all passed in the lower halves of consecutive registers when calling a variable-argument function.

The type 3 (used by the classic ABI for arg_ARGREAL, i.e., doubles) is reserved, and not used by this ABI.

5.2.2.3 64-Bit Integer Support Routines

An implementation must provide long long support routines (ASCII conversions, multiply, divide, and modulus operations, *etc*.) as specified in ISO/IEC 9899:1999. These include:

```
atoll           strtoll           strtoull           llabs
lldiv
```

5.2.2.4 APU-Handling Routines

An implementation must provide the processor-specific support routines in libc shown in Figure 5-3.

```
_ _get_apu_revision                    __get_apu_emul_revision
```

Figure 5-3. libc Required Routines

The mechanism for determining at run time which APUs are available on a processor involves the following function:

int _ _get_apu_revision(unsigned short)

This function takes a 16-bit identifier specifying the APU about which information is sought. It returns a signed 32-bit value.

- A positive value specifies the revision of the APU that is available.
- A negative value specifies that the operating system can provide emulation of the APU. The supported revision of the emulated APU corresponds to the absolute value of the return code.

- If the value is zero, the APU is not supported, either in hardware or through emulation by the operating system.

Applications can explicitly request information only about emulated support by using the following function:

int _ _ get_apu_emul_revision(unsigned short)

This function takes a 16-bit identifier specifying the APU about which information is sought. It returns a 32-bit value.

- The value can never be positive, as this implies hardware support.
- A negative value specifies that the operating system can provide emulation of the APU. The supported revision of the emulated APU corresponds to the absolute value of the return code.
- If the value is zero, the APU is not supported, either in hardware or through emulation by the operating system.

5.2.3 Processor-Specific Optional Routines

If long double support is provided, it should comply with ISO/IEC 9899:1999.

5.2.4 Optional Support Routines

In addition to the processor-specific routines specified above, libc may also contain the processor-specific support routines shown in Figure 5-4.

```
_q_lltoq        _q_qtoll        _q_qtoull       _q_ulltoq
_ _div64        _ _dtoll        _ _dtoull       _ _rem64
_ _udiv64       _ _urem64
```

Figure 5-4. libc Optional Support Routines

The following routines support software emulation of arithmetic operations for implementations that provide 64-bit signed and unsigned integer data types. In the descriptions below, the names long long (or signed long long) and unsigned long long are used to refer to these types. The routines employ the standard calling sequence described in Section 2.2, “Function Calling Sequence.” Descriptions are written from the caller’s point of view with respect to register usage and stack frame layout.

Note that the functions prefixed by _q_ below implement extended precision arithmetic operations. The following restriction applies to each of these functions:

If any floating-point exceptions occur, the appropriate exception bits in the SPEFSCR are updated; if the corresponding exception is enabled, the floating-point exception trap handler is invoked.

NOTE

The references in the following descriptions to a and b, where the corresponding arguments are pointers to long double quantities, refer to the values pointed to, not the pointers themselves.

long double _q_lltoq(long long a)
: This function converts the long long value of a to extended precision and returns the extended precision value.

long long _q_qtoll(const long double *a)
: This function converts the extended precision value of a to a signed long long by truncating any fractional part and returns the signed long long value.

unsigned long long _q_qtoull(const long double *a)
: This function converts the extended precision value of a to an unsigned long long by truncating any fractional part and returns the unsigned long long value.

long double _q_ulltoq(unsigned long long a)
: This function converts the unsigned long long value of a to extended precision and returns the extended precision value.

long long _ _div64(long long a, long long b)
: This function computes the quotient a/b, truncating any fractional part, and returns the signed long long result.

long long _ _dtoll(double a)
: This function converts the double precision value of a to a signed long long by truncating any fractional part and returns the signed long long value.

unsigned long long _ _dtoull(double a)
: This function converts the double precision value of a to an unsigned long long by truncating any fractional part and returns the unsigned long long value.

long long _ _rem64(long long a, long long b)
: This function computes the remainder upon dividing a by b and returns the signed long long result.

unsigned long long _ _udiv64(unsigned long long a, unsigned long long b)
: This function computes the quotient a/b, truncating any fractional part, and returns the unsigned long long result.

unsigned long long _ _urem64(unsigned long long a, unsigned long long b)
: This function computes the remainder upon dividing a by b and returns the unsigned long long result.

5.2.5 Software Floating-Point Emulation Support Routines

A high-level language processor, such as a compiler, may have a means of achieving floating-point arithmetic, comparisons, loads, and stores by generating software floating-point emulation (sfpe) code, rather than using PowerPC floating-point instructions. A language processor that supports sfpe code may support conversions between floating-point and 64-bit integer (for example, C's long long) data types. In sfpe code:

- Floating-point registers, the SPEFSCR (or EFSCR if only the scalar floating-point instruction set is implemented), and any PowerPC register bits that could cause a floating-point exception shall not be accessed.
- Floating-point single-precision scalars shall be passed the same as, be returned the same as, and have the same alignment as int scalars. Single-precision members of aggregates shall have the size and alignment of int members.
- Floating-point double-precision scalars shall be passed the same as, be returned the same as, and have the same alignment as long long scalars. Double-precision members of aggregates shall have the size and alignment of long long members.
- A caller of a function that takes a variable argument list shall not set condition register bit 6 to 1, since no arguments are passed in the floating-point registers.

The following restrictions shall apply to each of the sfpe support routines below, which are intended to be called by application sfpe code:

- The routines shall be sfpe code. for example, float and double in the descriptions mean sfpe single-precision and double-precision scalars, respectively, and no floating-point registers will be accessed.
- Floating-point arithmetic and comparisons by the routines shall be IEEE 754 conformant.
- Floating-point arithmetic and comparisons by the routines shall be performed as if all PowerPC floating-point exceptions have been disabled and shall not raise floating-point exceptions.

Conformant library support of sfpe code shall include all of routines in Figure 5-5 (routine interfaces are shown as C function prototypes).

```
_fp_round       _d_add          _d_cmp          _d_cmpe
_d_div          _d_dtof         _d_dtoi         _d_dtoq
_d_dtou         _d_feq          _d_fge          _d_fgt
_d_fle          _d_flt          _d_fne          _d_itod
_d_mul          _d_neg          _d_qtod         _d_sub
_d_utod         _f_add          _f_cmp          _f_cmpe
_f_div          _f_feq          _f_fge          _f_fgt
_f_fle          _f_flt          _f_fne          _f_ftod
_f_ftoi         _f_ftoq         _f_ftou         _f_itof
_f_mul          _f_neg          _f_qtof         _f_sub
_f_utof
```

Figure 5-5. SFPE Library Routines

int _fp_round(int rounding_mode)

This function shall set the rounding mode for sfpe library routines. If rounding_mode is 0, then round to nearest shall be requested; rounding_mode of 1 shall request round toward 0; rounding_mode of 2 shall request round toward positive infinity; rounding_mode of 3 shall request round toward negative infinity.

This function shall return the resulting rounding mode (0 for round to nearest, etc.) - which shall be rounding_mode if that rounding mode is supported by the sfpe library routines. Only round to nearest (This function returns 0) shall be required for conformance.

double _d_add(double a, double b)

This function shall return a + b computed to double-precision.

int _d_cmp(double a, double b)

This function shall perform an unordered comparison of the double-precision values of a and b and shall return an integer value that indicates their relative ordering as shown in Table 5-2.

Table 5-2. int _d_cmp(double a, double b) Relative Ordering

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2
a unordered with respect to b	3

int _d_cmpe(double a, double b)

This function shall perform an ordered comparison of the double-precision values of a and b and shall return an integer value that indicates their relative ordering, as shown in Table 5-3.

Table 5-3. int _d_cmpe(double a, double b) Relative Ordering

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2

double _d_div(double a, double b)

This function shall return a / b computed to double-precision.

float _d_dtof(double a)

This function shall convert the double-precision value of a to single-precision and shall return the single-precision value.

int _d_dtoi(double a)

This function shall convert the double-precision value of a to a signed integer by truncating any fractional part and shall return the signed integer value.

long double _d_dtoq(double a)

This function shall convert the double-precision value of a to extended precision and shall return the extended precision value.

unsigned int _d_dtou(double a)

This function shall convert the double-precision value of a to an unsigned integer by truncating any fractional part and shall return the unsigned integer value.

int _d_feq(double a, double b)

This function shall perform an unordered comparison of the double-precision values of a and b and shall return 1 if they are equal, and 0 otherwise.

int _d_fge(double a, double b)

This function shall perform an ordered comparison of the double-precision values of a and b and shall return 1 if a is greater than or equal to b, and 0 otherwise.

int _d_fgt(double a, double b)

This function shall perform an ordered comparison of the double-precision values of a and b and shall return 1 if a is greater than b, and 0 otherwise.

int _d_fle(double a, double b)

This function shall perform an ordered comparison of the double-precision values of a and b and shall return 1 if a is less than or equal to b, and 0 otherwise.

int _d_flt(double a, double b)

This function shall perform an ordered comparison of the double-precision values of a and b and shall return 1 if a is less than b, and 0 otherwise.

int _d_fne(double a, double b)

This function shall perform an unordered comparison of the double-precision values of a and b and shall return 1 if they are unordered or not equal, and 0 otherwise.

double _d_itod(int a)

This function shall convert the signed integer value of a to double-precision and shall return the double-precision value.

double _d_mul(double a, double b)

This function shall return a * b computed to double-precision.

double _d_neg (double a)

This function shall return -a.

double _d_qtod(const long double *a)

This function shall convert the extended precision value of a to double-precision and shall return the double-precision value.

double _d_sub(double a, double b)

This function shall return a - b computed to double-precision.

double _d_utod(unsigned int a)

This function shall convert the unsigned integer value of a to double-precision and shall return the double-precision value.

float _f_add(float a, float b)

This function shall return a + b computed to single-precision.

int _f_cmp(float a, float b)

This function shall perform an unordered comparison of the single-precision values of a and b and shall return an integer value that indicates their relative ordering, as described in Table 5-4.

Table 5-4. int _f_cmp(float a, float b) Relative Ordering

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2
a unordered with respect to b	3

int _f_cmpe(float a, float b)

This function shall perform an ordered comparison of the single-precision values of a and b and shall return an integer value that indicates their relative ordering, as shown in Table 5-5.

Table 5-5. int _f_cmpe(float a, float b) Relative Ordering

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2

float _f_div(float a, float b)

This function shall return a/b computed to single-precision.

int _f_feq(float a, float b)

This function shall perform an unordered comparison of the

single-precision values of a and b and shall return 1 if they are equal, and 0 otherwise.

int _f_fge(float a, float b)

This function shall perform an ordered comparison of the single-precision values of a and b and shall return 1 if a is greater than or equal to b, and 0 otherwise.

int _f_fgt(float a, float b)

This function shall perform an ordered comparison of the single-precision values of a and b and shall return 1 if a is greater than b, and 0 otherwise.

int _f_fle(float a, float b)

This function shall perform an ordered comparison of the single-precision values of a and b and shall return 1 if a is less than or equal to b, and 0 otherwise.

int _f_flt(float a, float b)

This function shall perform an ordered comparison of the single-precision values of a and b and shall return 1 if a is less than b, and 0 otherwise.

int _f_fne(float a, float b)

This function shall perform an unordered comparison of the single-precision values of a and b and shall return 1 if they are unordered or not equal, and 0 otherwise.

double _f_ftod(float a)

This function shall convert the single-precision value of a to double-precision and shall return the double-precision value.

int _f_ftoi(float a)

This function shall convert the single-precision value of a to a signed integer by truncating any fractional part and shall return the signed integer value.

long double _f_ftoq(float a)

This function shall convert the single-precision value of a to extended precision and shall return the extended precision value.

unsigned int _f_ftou(float a)

This function shall convert the single-precision value of a to an unsigned integer by truncating any fractional part and shall return the unsigned integer value.

float _f_itof(int a)

This function shall convert the signed integer value of a to single-precision and shall return the single-precision value.

float _f_mul(float a, float b)

This function shall return a * b computed to single-precision.

float _f_neg (float a)

This function shall return -a.

float _f_sub(float a, float b)

This function shall return a - b computed to single-precision.

float _f_utof(unsigned int a)

This function shall convert the unsigned integer value of a to single-precision and shall return the single-precision value.

Conformant library support of sfpe code may include the routines in Figure 5-6, which convert between floating-point and 64-bit integer data types (for example, C's long long), and shall include all of them if any is included.

`_d_dtoll`	`_d_dtoull`	`_d_lltod`	`_d_ulltod`
`_f_ftoll`	`_f_ftoull`	`_f_lltof`	`_f_ulltof`

Figure 5-6. SFPE Library Routines Supporting 64-bit Integer Data Types

long long _d_dtoll(double a)

This function shall convert the double-precision value of a to a signed long long by truncating any fractional part and shall return the signed long long value.

unsigned long long _d_dtoull(double a)

This function shall convert the double-precision value of a to an unsigned long long by truncating any fractional part and shall return the unsigned long long value.

double _d_lltod(long long a)

This function shall convert the signed long long value of a to double-precision and shall return the double-precision value.

double _d_ulltod(unsigned long long a)

This function shall convert the unsigned long long value of a to double-precision and shall return the double-precision value.

long long _f_ftoll(float a)

This function shall convert the single-precision value of a to a signed long long by truncating any fractional part and shall return the signed long long value.

unsigned long long _f_ftoull(float a)

This function shall convert the single-precision value of a to an unsigned long long by truncating any fractional part and shall return the unsigned long long value.

float _f_lltof(long long a)
This function shall convert the signed long long value of a to single-precision and shall return the single-precision value.

float _f_ulltof(unsigned long long a)
This function shall convert the unsigned long long value of a to single-precision and shall return the single-precision value.

5.2.6 Global Data Symbols

The libc library requires that some global external data objects be defined for the routines to work properly. In addition to the corresponding data symbols listed in the System V ABI, the symbol shown in Figure 5-7 must be provided in the system library on all ABI-conforming systems implemented with the PowerPC architecture. Declarations for the data objects listed below can be found in the Data Definitions section of this chapter.

```
_ _huge_val
```

Figure 5-7. libc Global External Data Symbols

5.2.7 Application Constraints

As described above, libc provides symbols for applications. In a few cases, however, an application is obliged to provide symbols for the library. In addition to the application-provided symbols listed in this section of the System V ABI, conforming applications on processors implementing the PowerPC architecture are also required to provide the following symbols:

extern _end;
This symbol refers neither to a routine nor to a location with interesting contents. Instead, its address must correspond to the beginning of a program's dynamic allocation area, called the 'heap.' Typically, the heap begins immediately after the data segment of the program's executable file.

extern const int _lib_version;
This variable's value specifies the compilation and execution mode for the program. If the value is zero, the program wants to preserve the semantics of older (pre-ANSI) C, where conflicts exist with ANSI. Otherwise, the value is nonzero, and the program wants ANSI C semantics.

5.3 System Data Interfaces

The only files changed from the classic PowerPC ABI are setjmp.h and sys/ucontext.h.

NOTE

Applications that use setjmp and ucontext interfaces are not portable between PowerPC ABIs.

The following may contain changes:

- float.h

The following files require changes:

- setjmp.h
- sys/ucontext.h

The Figure 5-8 and Figure 5-9 show the contents of changed files.

```
#define _GPR_JBLEN   18
#define _SIGJBLEN    40


typedef struct {
    int stackpointer;
    int lr;
    long long r2;
    long long gprs_nonvolatile[_GPR_JBLEN];
} jmp_buf[1];


typedef int sigjmp_buf[_SIGJBLEN];
```

Figure 5-8. <setjmp.h> Contents

```
/* Original portions of ucontext.h above this #define remains unchanged, although some
changes may be required to track SPE state. */
#define  NGREG      82

typedef  int        gpr32reg_t;
typedeflong longgpr64reg_t;
typedef  greg_t     gprregset_t[NGREG];

struct regs {
```

```
        gpr64reg_t r_r0;
        gpr64reg_t r_r1;
        gpr64reg_t r_r2;
        gpr64reg_t r_r3;
        gpr64reg_t r_r4;
        gpr64reg_t r_r5;
        gpr64reg_t r_r6;
        gpr64reg_t r_r7;
        gpr64reg_t r_r8;
        gpr64reg_t r_r9;
        gpr64reg_t r_r10;
        gpr64reg_t r_r11;
        gpr64reg_t r_r12;
        gpr64reg_t r_r13;
        gpr64reg_t r_r14;
        gpr64reg_t r_r15;
        gpr64reg_t r_r16;
        gpr64reg_t r_r17;
        gpr64reg_t r_r18;
        gpr64reg_t r_r19;
        gpr64reg_t r_r20;
        gpr64reg_t r_r21;
        gpr64reg_t r_r22;
        gpr64reg_t r_r23;
        gpr64reg_t r_r24;
        gpr64reg_t r_r25;
        gpr64reg_t r_r26;
        gpr64reg_t r_r27;
        gpr64reg_t r_r28;
        gpr64reg_t r_r29;
        gpr64reg_t r_r30;
        gpr64reg_t r_r31;
        gpr64reg_t r_acc;     /* SPE Accumulator */

        gpr32reg_t r_cr;       /* Condition register */
        gpr32reg_t r_lr;       /* Link register */
        gpr32reg_t r_pc        /* User PC (Copy of SRR0) */
        gpr32reg_t r_msr;      /* Saved MSR (Copy of SRR1) */
        gpr32reg_t r_ctr;      /* Count register */
        gpr32reg_t r_xer;      /* Integer exception register */
        gpr32reg_t r_spefscr;  /* SPE/Floating-point status and control register */};/*
struct fpu is deprecated. */

typedef struct {
        gprregset_t            gregs;
};

/* Portions of ucontext.h below this point remains unchanged. */
```

Figure 5-9. <ucontext.h> Contents

Appendix A
Differences Between ABIs

This appendix summarizes some of the main differences between the “classic” System V Release 4 PowerPC ABI and this e500 ABI. Several of the changes were to pull in features of the PowerPC Embedded ABI (EABI), and are discussed as such.

This appendix is not authoritative: there are several minor changes that are not reflected here.

A.1 Introduction

- e500ABI contains a compatibility summary
- e500ABI contains many more references to appropriate, updated documents

A.2 Software Installation

- e500ABI removed this chapter

A.3 Low-Level System Information

- e500ABI adds long long types to the Scalar Types table
- e500ABI clarifies ABI-compliant handling of doubles and long doubles
- e500ABI adds the `__ev64_opaque__` type, for SPE support
- e500ABI adds discussion about upper 32 bits of the GPRs
- e500ABI allows r2 to be used as an `sdata2` pointer
- e500ABI clarified register usage by adding “dedicated” and “limited-access” volatility
- e500ABI adds discussion about SPEFSCR register bits
- e500ABI adds 8-byte-aligned 64-bit general register save area to the stack frame
- e500ABI discusses saving the upper words of the GPRs
- e500ABI specifies parameter passing changes for `__ev64_opaque__`, float, and double types
- e500ABI adds a summary of argument and return value handling
- e500ABI adds stack frame examples

- e500ABI marks the Operating System Interface as optional
- e500ABI changes the recommended program base to 0x1000_0000
- e500ABI marks the Exception Interface as optional
- e500ABI specifies SPE exceptions should map to SIGILL
- e500ABI specifies that cache-locking exceptions should map to SIGSEGV
- e500ABI marks Process Initialization as optional
- e500ABI specifies SPEFSCR initial value
- e500ABI specifies register saving and restoring for 32-bit and 64-bit register usage
- e500ABI specifies DWARF register numbers for the upper 32-bits of the GPRs, and also specifies the ACC, SPEFSCR, and PMRs, as well as AltiVec VRs and DCRs, even though the latter do not apply to the e500 core

A.4 Object Files

- e500ABI adds support for the EABI special sections (.sdata2/.sbss2, .sdata0/.sbss0, .seginfo), but prefixes them with .PPC.EMB to comply with the latest System V ABI
- e500ABI adds an .PPC.EMB.apuinfo section, to allow specification of expected APUs
- e500ABI has removed tag discussions, as DWARF can now provide equivalent information
- e500ABI adds relocations for SPE instructions
- e500ABI adds several EABI relocations: SDA_I16, SDA2_I16, SDA21, MRKREF, BIT_FLD, RELSDA. Note that all of these are now prefixed with R_PPC_EMB, to comply with the latest System V ABI

A.5 Program Loading and Dynamic Linking

- e500ABI marks this chapter as “extended conformance”

A.6 Libraries

- e500ABI has reorganized much of this chapter, as the System V ABI and modern C environments now incorporate much of the additional functionality specified in the classic PowerPC ABI
- e500ABI specifies that the save and restore routines are required to be in `libc`.
- e500ABI specifies APU-handling routines
- e500ABI adds the Software Floating-Point Emulation (SFPE) routines from the EABI
- e500ABI specifies some changes to the System Data Interfaces to support SPE

INDEX

P

R

S