PowerPC Embedded Application Binary Interface

32-Bit Implementation

Version	Description	Date
1.0	Initial Release	01/10/95

for information contact:

Stephen Sobek
Microcontroller Technologies Group
Motorola
6501 William Cannon Drive West
Mail Stop OE45
Austin, TX 78735
steve@avar.sps.mot.com

Kevin Burke
Technology Products
International Business Machines Corporation
3039 Cornwallis Road
Mail Stop H83/061
Research Triangle Park, NC 27709
kevin_burke@vnet.ibm.com

Important Notice to Users

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any information, product, or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, *including implied warranties of merchantability or fitness for a particular purpose*. Motorola makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

Trademarks

The following trademarks apply to this document:

PowerPC and IBM are trademarks of International Business Machines Corporation.

Motorola is a trademark of Motorola, Inc.

UNIX is a registered trademark of Novell, Inc. in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Foreword

The PowerPC Embedded Application Binary Interface, or EABI, was created to meet the unique needs of PowerPC embedded applications, specifically minimizing memory usage while maintaining high performance. The EABI was developed by an industry-wide working group consisting of PowerPC chip manufacturers, software tool vendors, and end users.

The EABI defines a set of conventions intended to afford interoperability between conforming software components (e.g., compilers, debuggers, assembly language code). These conventions are optimized for embedded applications, which typically differ from desktop applications in at least one of the following ways:

- ROM based
- real-time oriented
- memory constrained
- single purpose application

The EABI was created with the explicit goal of being as close as possible to an existing desktop ABI. While several alternatives were considered (e.g., PowerOpen ABI), the one chosen was the Unix System V Release 4 (SVR4) ABI for PowerPC. This choice makes it possible to use SVR4 functions in an EABI environment without recompilation. The two ABI's are sufficiently close that software tool vendors can support both ABI's with a single set of tools.

Embedded programs that conform to the EABI gain efficiency in space and time by using the following features:

- minimized stack usage
- relaxed alignment restrictions, optimizing memory usage
- small data areas for RAM data, read-only data, and data around address zero. These reduce code size and improve data access time.

Acknowledgements

Members of the PowerPC Embedded Application Binary Interface Working Group who devoted substantial time and effort to create this standard included:

Tom Barrett	Mark Edwards	Ken Greenberg	Steve Sobek
Kevin Burke	Tomas Evensen	David Layman	Budi Sutardja
Felix Burton	Chris Ford	Steve Mihalik	Fred Viles
Thomas Collopy	Craig Franklin	Rebecca Moeller	Alex Wu
Bill DeStein	Tony Goelz	Jack Patteeuw	Steven Zucker

The working group is grateful for the efforts made by all participants.

References

ANSI X3.159-1989, American National Standard for Information Systems - Programming Language - C, American National Standards Institute, 1989.

DWARF Debugging Information Format, Revision: Version 1.1.0, UNIX International, Programming Languages SIG, October 6, 1992.

DWARF Debugging Information Format, Revision: Version 2.0.0, Industry Review Draft, UNIX International, Programming Languages SIG, July 27, 1993.

PowerPC Microprocessor Family: The Programming Environments, Motorola, Motorola document MPCFPE/AD, IBM Microelectronics document MPRPPCFPE-01, 1994.

System V Application Binary Interface, Third Edition, UNIX System Laboratories, 1994 (ISBN 0-13-100439-5).

System V Application Binary Interface, PowerPC Processor Supplement, Sun Microsystems and IBM, 1995.

Table of Contents

CHAPTER 1

PowerPC Embedded Application Binary Interface

Introduction and Overview

The PowerPC Embedded Application Binary Interface (EABI) defines a system interface for compiled and assembled embedded application programs that will run on embedded 32-bit implementations of the PowerPC architecture.

Terminology

The word "shall" in this document denotes a characteristic or behavior that is mandatory for an EABI-conforming entity, such as an application program or compiler. The word "may" denotes a characteristic or behavior that is explicitly permitted for an EABI-conforming entity.

Relationship to UNIX System V Release 4

Except as noted in this document, the EABI adopts the specifications found in those editions of the *System V Application Binary Interface* and the *System V Application Binary Interface*, *PowerPC Processor Supplement* that are listed in the references.

Base and Extended Conformance

This document defines two levels of conformance - **base EABI conformance** and **extended EABI conformance**.

Base EABI conformance shall consist of conforming to all requirements that are not labeled *EXTENDED*.

Extended EABI conformance shall consist of conforming to all requirements, including those labeled *EXTENDED*, with a requirement labeled *EXTENDED* superseding any conflicting base conformance requirements. E.g., the extended EABI conformance requirement that a static linker support all SVR4 ABI relocation types supersedes the base EABI conformance requirement that the relocation types in Table 4-1 are not supported.

Definitions and Notation

The following definitions and notation are used throughout this document:

- **SVR4** is the UNIX System V Release 4 operating system.
- **SVR4 ABI** is the UNIX System V Release 4 Application Binary Interface, including the PowerPC processor supplement.
- Numbers are decimal unless specified in the following way:
  - **0x*nnnn***, where ***nnnn*** is a non-empty sequence of hexadecimal digits, denotes a hexadecimal number whose value, expressed in hexadecimal, is ***nnnn***.
- A **byte** is 8 bits.
- A **word** is 32 bits (4 bytes).
- **GPR*n***, where ***n*** is a number, denotes PowerPC General Purpose Register ***n***.

Future Directions

Characteristics and behaviors mandated by this version of the EABI shall continue to be mandated indefinitely except where this document explicitly states otherwise. All mandates that might be withdrawn or altered in the next edition of the EABI are preceded by **WARNING:** and appear in bold face type.

CHAPTER 2 Software Installation

Unlike the SVR4 ABI, the EABI shall not have required physical media for distribution of EABI-conforming application software, a required software format (such as a continuous data stream) on the physical media, a required layout of files on the physical media, or a required format or interpretation of installation data files.

CHAPTER 3

Low-Level System Information

SVR4 ABI Execution Support

A conforming entity, such as an application or a static linker, shall not have requirements pertaining to:

- dynamic linking
- global offset tables
- procedure linkage tables
- shared objects

EXTENDED A conforming static linker, dynamic linker, or high-level language processor (such as a compiler) shall implement all of the following in a way that conforms to the SVR4 ABI, and a conforming application that uses any of the following shall do so in a way that conforms to the SVR4 ABI:

- dynamic linking
- global offset tables

- procedure linkage tables
- shared objects

Machine Interface

Processor Architecture

Unlike the SVR4 ABI, which only allows non-privileged PowerPC instructions, an EABI-conforming application program shall be allowed to use any privileged or optional instruction defined by the PowerPC architecture, with the following exceptions. As in the SVR4 ABI, Fixed-Point Load and Store Multiple instructions and the Fixed Point Move Assist instructions shall not be allowed in EABI-conforming Little Endian applications.

Data Representation

Fundamental Types

Unlike the SVR4 ABI, the alignment of a long double shall be 8 bytes (doubleword), although the size of long double shall be 16 bytes.

Aggregates and Unions

Unlike the SVR4 ABI, an array, structure or union containing a long double shall start aligned on an 8 byte boundary. However, as in the SVR4 ABI, a long double member within a structure or union shall start at the lowest available offset aligned on a 16 byte boundary, and the size of a structure or union with a long double member shall be a multiple of 16 bytes.

Function Calling Sequence

Registers

Unlike the SVR4 ABI, GPR2 shall not be reserved for system use, but shall instead be dedicated to contain the **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 GPR2.

The Stack Frame

Unlike the SVR4 ABI, the stack pointer (GPR1) shall maintain 8-byte alignment, from initialization through all routine calls and dynamic stack space allocation.

Operating System Interface

Unlike the SVR4 ABI, an EABI-conforming entity shall not have operating system interface requirements.

Exception Interface

Unlike the SVR4 ABI, an EABI-conforming entity shall not have exception interface requirements.

Process Initialization

Unlike the SVR4 ABI, an EABI-conforming entity shall not have process initialization requirements.

Coding Examples

Data Objects

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 .sdata2 and .sbss2 is within a signed 16-bit offset of _SDA2_BASE_'s value (see "Special Sections" on page 11).

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

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

1. Sections .sdata, .sbss, and .sdata2 shall contain at least the following:
   - Entries for those defined variables that are globally visible scalars of size 8 or fewer bytes and whose values will not be changed outside of the program (which excludes C variables that are volatile).

- Every such defined variable whose initial value is explicitly non-zero and might be changed by the program shall have a .sdata entry that represents the variable.
- Every such defined variable whose value is initially 0 and might be changed shall have a .sbss entry or a .sdata entry that represents the variable.
- If the ELF object file generated is not intended to be part of a shared object file, every such variable whose value cannot be changed by the program (such as a C variable that is const but not volatile) shall have a .sdata2 entry that represents the variable; otherwise, such constant variables shall have .sdata or .sbss entries, as appropriate.

• Entries produced by the static linker's resolution of relocation types (see "Relocation Types" on page 15).

2. The only external variables accessed by the generated code as .sdata, .sbss, .sdata2, .sbss2, .PPC.EMB.sdata0, or .PPC.EMB.sbss0 entries shall be as follows:

- External variables that are scalars of 8 or fewer bytes, whose values might be changed by the program, and whose values will not be changed outside of the program, shall be accessed as .sdata or .sbss entries. So the address of such a variable will be within a 16-bit signed offset of _SDA_BASE_, which in a shared object file is the same value as _GLOBAL_OFFSET_TABLE_, and otherwise is loaded in GPR13 by a conforming application.
- When the object file is not to be part of a shared object file, external variables that are scalars of 8 or fewer bytes, whose values cannot be changed by the program, and whose values will not be changed outside of the program, shall be accessed as .sdata2 or .sbss2 entries. In a shared object file, those constant external variables shall be accessed as .sdata or .sbss entries. So the address of such a variable, when not in a shared object file, will be within a 16-bit signed offset of _SDA2_BASE_, which is loaded into GPR2 by a conforming application.

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

```
        int                    i_sdata          = 1;
        const int              i_sdata2         = 2;
        int                    i_sbss_or_sdata;
        short                  s_sbss_or_sdata  = 0;
extern  double                 d_sdata_or_sbss;
extern  const double           d_sdata2;

extern  double                 d_any_sdata_or_sbss[50];
extern  const float            f_any_sdata_or_sbss[200];
extern  union my_union         u_any_sdata_or_sbss;
extern  const volatile float   cvf_any_sdata_or_sbss;

        int                    i_any_sdata[100]   = { 3 };
static  struct my_struct       s_any_sdata        = { 4, 6 };
        volatile const float   vcf_any_sdata[5]   = { 5 };

        int                    i_any_sbss_or_sdata[100];
static  struct my_struct       s_any_sbss_or_sdata;
        volatile const float   vcf_any_sbss_or_sdata[25];
```

If the code fragment defines all globally visible variables, a C compiler when conforming to the rules above would place i_sdata in .sdata, i_sdata2 in .sdata2, and i_sbss_or_sdata and s_sbss_or_sdata in either .sbss or .sdata, while at the same time generating code that accesses external variable d_sdata_or_sbss using an offset relative to the value of _SDA_BASE_ (which is in GPR13), accesses d_sdata2 using an offset relative to _SDA2_BASE_ (which is in GPR2), and does not access any other external variables as .sdata, .sbss, .sdata2, .sbss2, .PPC.EMB.sdata0, or .PPC.EMB.sbss0 entries.

Except when conforming to the rules above, a conforming C compiler could generate code accessing external variables d_any_sdata_or_sbss, f_any_sdata_or_sbss, u_any_sdata_or_sbss, and cvf_any_sdata_or_sbss relative to the value of _SDA_BASE_, relative to _SDA2_BASE_, or relative to address 0. (Although section .sdata2 generally is used to hold only constant data.)

Even when conforming to the rules above, as long as section size restrictions are met, any variables or unnamed data can be in .sdata, .sdata2 or .PPC.EMB.sdata0, and any variables or unnamed data that are initially 0 can be in .sbss, .sbss2, or .PPC.EMB.sbss0. So a conforming compiler might place i_any_sdata, s_any_sdata, and vcf_any_sdata in .sdata, .sdata2, or .PPC.EMB.sdata0. A compiler might be able to put i_any_sbss_or_sdata, s_any_sbss_or_sdata, or vcf_any_sbss_or_sdata in .sbss, .sbss2, or .PPC.EMB.sbss0, or even explicitly initialize those variables with zeroes and place them in .sdata, .sdata2, or .PPC.EMB.sdata0. Finally, when not conforming to the rule above that restricts their placement, i_sdata, i_sdata2, i_sbss_or_sdata and s_sbss_or_sdata could be placed in other appropriate sections (such as putting i_sdata in .PPC.EMB.sdata0).

Dynamic Stack Space Allocation

Unlike the SVR4 ABI, the stack pointer (GPR1) shall maintain 8-byte alignment.

DWARF Definition

Unlike the SVR4 ABI, which does not define a debug format, the EABI adopts DWARF as the debugging information format for EABI-conforming applications. An EABI-conforming application that uses a debug format shall use either DWARF Version 1.1.0 or DWARF Version 2.0.0.

WARNING: At some future time, use of DWARF Version 1.1.0 by EABI-conforming applications might be disallowed.

DWARF Register Number Mapping

An EABI-conforming application shall use for both DWARF Version 1.1.0 and DWARF Version 2.0.0 the DWARF register numbering specified in the SVR4 ABI.

Address Class Codes

An EABI-conforming application shall use for both DWARF Version 1.1.0 and DWARF Version 2.0.0 the address class codes specified in the SVR4 ABI.

CHAPTER 4 Object Files

Object File Processing

An EABI-conforming static linker shall accept as input EABI-conforming and SVR4-conforming relocatable object files, and it shall produce EABI-conforming object files.

ELF Header

Machine Information

The ELF header's e_flags member of an EABI-conforming object file shall have bit 0x80000000 set, defined as the name EF_PPC_EMB.

Sections

Special Sections

In addition to the special sections in the SVR4 ABI, an EABI-conforming object file shall be allowed to contain the special sections described below. The SVR4 ABI has reserved for this document the section names .sdata2, .sbss2, and those beginning with the string ".PPC.EMB.".

The special section .sdata2 is 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. The special section .sbss2 is intended to hold writable small data that contribute to the program memory image and whose initial values are 0.

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

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

In the section header for .sdata2:

- sh_type shall be SHT_PROGBITS
- sh_flags shall be either SHF_ALLOC or SHF_ALLOC + SHF_WRITE
- sh_link shall be SHN_UNDEF
- sh_addralign shall be the maximum alignment required by any data item in .sdata2
- sh_info and sh_entsize shall be 0

If a static linker creates a .sdata2 section that combines a .sdata2 section whose sh_flags is SHF_ALLOC with a .sdata2 section whose sh_flags is SHF_ALLOC + SHF_WRITE, then the resulting .sdata2 section's sh_flags value shall be SHF_ALLOC + SHF_WRITE.

In the section header for .sbss2:

- sh_type shall be SHT_NOBITS
- sh_flags shall be SHF_ALLOC + SHF_WRITE
- sh_link shall be SHN_UNDEF
- sh_addralign shall be the maximum alignment required by any data item in .sbss2
- sh_info and sh_entsize shall be 0

The special section .PPC.EMB.sdata0 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. The special section .PPC.EMB.sbss0 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.

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

In the section header for .PPC.EMB.sdata0:

- sh_type shall be SHT_PROGBITS
- sh_flags shall be SHF_ALLOC + SHF_WRITE
- sh_link shall be SHN_UNDEF
- sh_addralign shall be the maximum alignment required by any data item in .PPC.EMB.sdata0
- sh_info and sh_entsize shall be 0

In the section header for .PPC.EMB.sbss0:

- sh_type shall be SHT_NOBITS
- sh_flags shall be SHF_ALLOC + SHF_WRITE
- sh_link shall be SHN_UNDEF
- sh_addralign shall be the maximum alignment required by any data item in .PPC.EMB.sbss0
- sh_info and sh_entsize shall be 0

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.

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_type shall be SHT_PROGBITS
- 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 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 the following:

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.

Relocation

Relocation Types

A static linker shall support all SVR4 ABI relocation types except for those listed in Table 4-1.

EXTENDED A static linker shall support all SVR4 ABI relocation types, including those in Table 4-1, and a dynamic linker shall support all SVR4 ABI relocation types appropriate to dynamic linking.

TABLE 4-1. SVR4 ABI Relocation Types for Extended Conformance

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

Table 4-2 lists the new relocation types defined by the EABI. The heading **Check** denotes whether the link shall fail if the value computed does not fit in the allocated bits.

A static linker shall support all relocation types in Table 4-2 and shall not accept a relocation entry whose relocation type is not defined in either Table 4-2 or the SVR4 ABI.

EXTENDED A dynamic linker shall not process a relocation entry whose relocation type is not defined in either Table 4-2 or the SVR4 ABI.

The relocatable fields of EABI relocation types shall have no alignment restrictions. Fields in Table 4-2 shall have the following meanings (since fields can be unaligned, names start with ‘u’):

- uword32 32-bit field occupying 4 bytes
- ulow21 21-bit field occupying the least significant bits of the 24-bit field pointed to by the relocation entry
- uhalf16 16-bit field occupying 2 bytes (e.g., Add Immediate instruction’s immediate field)

Calculations in Table 4-2 use the following symbols:

- A relocation entry’s r_addend field value
- S address (value) of the symbol whose index is in the relocation entry’s r_info field
- T offset from _SDA_BASE_ to where in .sdata the static linker placed the address of the symbol whose index is in r_info. See R_PPC_EMB_SDAI16 description below.
- U offset from _SDA2_BASE_ to where in .sdata2 the static linker placed the address of the symbol whose index is in r_info. See R_PPC_EMB_SDA2I16 description below.
- V offset to the symbol whose index is in r_info from the start of that symbol’s containing section
- W address of the start of the section containing the symbol whose index is in r_info

Calculations in Table 4-2 use the following notation:

- “+” and “-” denote 32-bit modulus addition and subtraction, respectively. “>>” 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 whose names contain the string “14” or the string “16”, the upper 17 bits of the value computed before shifting must all be the same.
- #hi(value) and #lo(value) denote the most and least significant 16 bits, respectively, of the value. That is, #lo(x) = (x & 0xFFFF) and #hi(x) = ((x >> 16) & 0xFFFF). #ha(value) compensates for treating #lo() as a signed number. #ha(x) = (((x >> 16) + ((x >> 15) & 0x1)) & 0xFFFF).

TABLE 4-2. EABI Relocation Types

Relocation Type	Value	Field	Check	Calculation
R_PPC_EMB_NADDR32	101	uword32	N	(A - S)
R_PPC_EMB_NADDR16	102	uhalf16	Y	(A - S)
R_PPC_EMB_NADDR16_LO	103	uhalf16	N	#lo(A - S)
R_PPC_EMB_NADDR16_HI	104	uhalf16	N	#hi(A - S)
R_PPC_EMB_NADDR16_HA	105	uhalf16	N	#ha(A - S)
R_PPC_EMB_SDAI16	106	uhalf16	Y	T
R_PPC_EMB_SDA2I16	107	uhalf16	Y	U
R_PPC_EMB_SDA2REL	108	uhalf16	Y	S + A - _SDA2_BASE_
R_PPC_EMB_SDA21	109	ulow21	N	See below
R_PPC_EMB_MRKREF	110	none	N	See below
R_PPC_EMB_RELSEC16	111	uhalf16	Y	V + A
R_PPC_EMB_RELST_LO	112	uhalf16	N	#lo(W + A)
R_PPC_EMB_RELST_HI	113	uhalf16	N	#hi(W + A)
R_PPC_EMB_RELST_HA	114	uhalf16	N	#ha(W + A)
R_PPC_EMB_BIT_FLD	115	uword32	Y	See below
R_PPC_EMB_RELSDA	116	uhalf16	Y	See below

R_PPC_EMB_SDAI16 This instructs a static 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.

R_PPC_EMB_SDA2I16 This instructs a static linker to create a 4-byte, word aligned, entry in the .sdata2 section containing the address of the symbol whose index is in r_info. At most one such implicit .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 3 bits at the address pointed to by the relocation entry shall be left unchanged. If the symbol whose index is in r_info is contained in .sdata or .sbss, then a static linker shall place in the next most significant 5 bits the value 13 (for GPR13); if the symbol is in .sdata2 or .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 .sdata2 or .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 static linker shall not apply a relocation action to a location because of this type. This relocation type is used to prevent a static 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 static 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 static 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 .sdata2 or .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 5

Program Loading and Dynamic Linking

Program Loading

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

Program Interpreter

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

CHAPTER 6 Libraries

SVR4 Library Routines

The chapter "LIBRARIES" in the *System V Application Binary Interface*, *PowerPC Processor Supplement* describes optional and required routines for the SVR4 libsys and libc libraries. An EABI-conforming library routine whose name matches the name of a routine described in chapter "LIBRARIES" (__va_arg, __dtou, and the routines in Tables 6-1 through 6-5) shall have the interface and semantics described in the SVR4 ABI. If any routine in Table 6-1, 6-2, 6-3, 6-4, or 6-5 is supported, then all other routines in the same table shall be supported.

TABLE 6-1. SVR4 ABI libsys 16-Byte Float Routines

_q_add	_q_cmp	_q_cmpe	_q_div
_q_dtoq	_q_feq	_q_fge	_q_fgt
_q_fle	_q_flt	_q_fne	_q_itoq
_q_mul	_q_neg	_q_qtod	_q_qtoi
_q_qtos	_q_qtou	_q_sqrt	_q_stoq
_q_sub	_q_utoq		

TABLE 6-2. SVR4 ABI libsys 8-Byte Integer Arithmetic Routines

__div64	__dtoll	__dtoull	__rem64
__udiv64	__urem64		

TABLE 6-3. SVR4 ABI libsys 8-Byte Integer to/from 16-Byte Float Routines

_q_lltoq	_q_qtoll	_q_qtoull	_q_ulltoq

TABLE 6-4. SVR4 ABI libc 8-Byte Integer to/from String Routines

atoll	lltostr	strtoll	strtoull
ulltostr	wstoll		

TABLE 6-5. SVR4 ABI libc 8-Byte Integer Arithmetic Routines

llabs	lldiv

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 (e.g., C's long long) data types. In sfpe code:

- Floating point registers, the FPSCR, 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 long int scalars. Single precision members of aggregates shall have the size and alignment of long 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. E.g., 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 Table 6-6 (routine interfaces are shown as C function prototypes).

TABLE 6-6. SFPE Library Routines

`_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`

`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:

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:

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:

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:

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 Table 6-7, which convert between floating point and 64-bit integer data types (e.g., C's long long), and shall include all of them if any is included.

TABLE 6-7. SFPE Library Routines Supporting 64-bit Integer Data Types

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

`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.

APPENDIX A EABI Summary

Table A-1 lists topics that are addressed differently in the EABI and the SVR4 ABI. Table A-2 lists topics in the EABI that are not in the SVR4 ABI. Table A-3 lists the additional requirements for EABI extended conformance over EABI base conformance. Table A-4 lists EABI topics for which support is optional. Table A-5 lists SVR4 ABI topics for which there are no EABI requirements. Page numbers refer to EABI pages.

TABLE A-1. Different in EABI and SVR4 ABI

Page	Topic	EABI	SVR4 ABI
5	PowerPC instruction set use	privileged and optional instructions allowed	privileged and optional instructions not allowed
5	long double scalar	8-byte aligned	16-byte aligned
5	struct/union containing long double	starts 8-byte aligned	starts 16-byte aligned
5	GPR2 usage	dedicated pointer to .sdata2 and .sbss2 sections	reserved for system use
6	stack pointer value	multiple of 8 (stack is 8-byte aligned)	multiple of 16 (stack is 16-byte aligned)
6	language processor (e.g., compiler) support of .sdata and .sbss sections	some required capabilities	only optional capabilities
9	dynamic stack allocation (e.g., by alloca)	8-byte aligned	16-byte aligned
15	SVR4 relocation types	linker support of certain types optional, except in extended conformance (See Table A-3.)	linker support of all types required
20	libsys and libc library routines	support optional (See Table A-4.)	some required and some optional routines

TABLE A-2. EABI Unique

Page	Topic	Description
6	symbol _SDA2_BASE_	value within 16-bit signed offset of all .sdata2/.sbss2 bytes
9	debug format	DWARF Version 1.1.0 or DWARF Version 2.0.0
11	.sdata2 and .sbss2 sections	small data area (64K or fewer bytes) pointed to by GPR2; some required language processor support
12	.PPC.EMB.sdata0 and .PPC.EMB.sbss0 sections	small data area within 16-bit signed offset of address 0
13	.PPC.EMB.seginfo	section created by static linker to name segments or declare one segment is ROM copy of a second; support for creation optional, and creating linker may support just naming, just ROM copies, or both (See Table A-4.)
15	EABI relocation types	several required relocation types
21	software floating point emulation (sfpe)	optionally supported language processor code generation restrictions and library routines for achieving floating point without using native PowerPC floating point; sfpe library support requires all single and double precision routines, and, if any 64-bit integer routine is supported, all such routines must be supported (See Table A-4.)

TABLE A-3. EABI Extended Conformance

Page	Description
4	dynamic linking as in SVR4 ABI
4	global offset tables as in SVR4 ABI
4	procedure linkage tables as in SVR4 ABI
4	shared objects as in SVR4 ABI
14	static linker supporting optional ROM copy segments ensures that relocation entries resolved by dynamic linkers do not refer to locations in a ROM copy segment
15	linker support of all SVR4, as well as all EABI, relocation types

TABLE A-4. EABI Optional Support

Page	Description
14	static linker creation of .PPC.EMB.seginfo, and whether a linker supporting creation allows only segment naming, only ROM copy segments, or both.
21	language processor (e.g. compiler) generation of software floating point emulation (**sfpe**) code, and whether such generation includes conversions between floating point and 64-bit integer data types
27	inclusion in libraries that support sfpe code of routines that convert between floating point and 64-bit integer data types

TABLE A-5. SVR4 ABI Unique

exception interface
operating system interface
process initialization
program interpreter
program loading
software distribution and installation