64-bit PowerPC ELF Application Binary Interface Supplement

Ian Lance Taylor
Zembu Labs
Kristin
Thomas

64-bit PowerPC ELF Application Binary Interface Supplement

by and Ian Lance Taylor
Kristin
Thomas

&specversion; Edition
Published &specdate;

Revision History

Revision 1.1 Revised by: David Edelsohn, IBM Research

Revision 1.2 Revised by: Swox AB

Revision 1.3 Revised by: David Edelsohn and Mark Mendell, IBM

Revision 1.4 Revised by: Alan Modra, IBM

Table of Contents

Chapter 1. Introduction

ELF defines a linking interface for compiled application programs. ELF is described in two parts. The first part is the generic System V ABI. The second part is a processor specific supplement.

This document is the processor specific supplement for use with ELF on 64-bit gPowerPC® processor systems.

This document is not a complete System V Application Binary Interface Supplement, because it does not define any library interfaces.

In the 64-bit gPowerPC Architecture™, a processor can run in either of two modes: big-endian mode or little-endian mode. (See the Section called *Byte Ordering* in Chapter 3g.) Accordingly, this ABI specification really 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. The 64-bit PowerPC ELF ABI is not the same as the 32-bit PowerPC ELF ABI, nor is it a simple extension. A system which supports the 64-bit PowerPC ELF ABI may, but need not, support the 32-bit PowerPC ELF ABI.

The 64-bit PowerPC ELF ABI is intended to use the same structure layout and calling convention rules as the 64-bit PowerOpen ABI.

How to Use the 64-bit PowerPC EWLF ABI Supplement

While the generic System V ABI is the prime reference document, this document contains 64-bit PowerPC processor-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 the book titled *IBM PowerPC User Instruction Set Architecture*, all of which should be considered part of this 64-bit PowerPC Processor ABI Supplement and just as binding as the requirements and data it explicitly includes.

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

- *System V Interface Definition*, Issue 3.
- *The PowerPC Architecture: A Specification for A New Family of RISC Processors*. International Business Machines (IBM). San Francisco: Morgan Kaufmann, 1994.
- *DWARF Debugging Information Format, Revision: Version 2.0.0* , July 27, 1993. UNIX International, Program Languages SIG.
- *The [32-bit] PowerPC Processor Supplement*, Sun Microsystems, 1995.
- *The 64-bit AIX ABI*
- *The PowerOpen ABI*

Chapter 2. Software Installation

Physical Distribution Media and Formats

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

Chapter 3. Low Level System Information

Machine Interface

Processor Architecture

The PowerPC Architecture: A Specification for A New Family of RISC Processors defines the 64-bit PowerPC Architecture. Programs intended to execute directly on the processor use the 64-bit PowerPC instruction set, and the instruction encodings and semantics of the architecture.

An application program can assume that all instructions defined by the architecture that are neither privileged nor optional exist and work as documented. However, the "Fixed-Point Move Assist" instructions are not available in little-endian implementations. In little-endian mode, these instructions always cause alignment exceptions in the 64-bit PowerPC Architecture; in big-endian mode they are usually slower than a sequence of other instructions that have the same effect.

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 64-bit PowerPC Architecture, or additional non-64-bit-PowerPC instructions or capabilities. Programs that use those instructions or capabilities do not conform to the 64-bit PowerPC ABI; executing them on machines without the additional capabilities gives undefined behavior.

Data Representation

Byte Ordering

The architecture defines an 8-bit byte, a 16-bit halfword, a 32-bit word, a 64-bit doubleword, and a 128-bit quadword. Byte ordering defines how the bytes that make up halfwords, words, doublewords, 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 64-bit PowerPC processor family supports 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. The following figures illustrate the 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: In the 64-bit PowerPC Architecture documentation, the bits in a word are numbered from left to right (MSB to LSB), and figures usually show only the big-endian byte order.

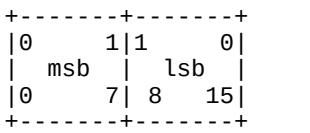


Figure 3-1. Bit and Byte Numbering in Halfwords

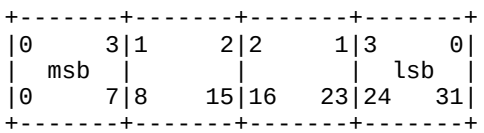


Figure 3-2. Bit and Byte Numbering in Words

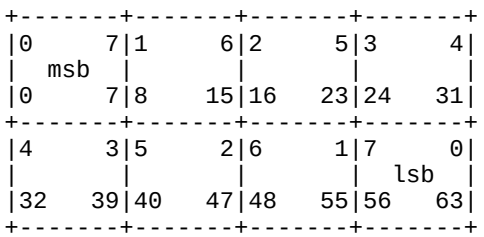


Figure 3-3. Bit and Byte Numbering in Doublewords

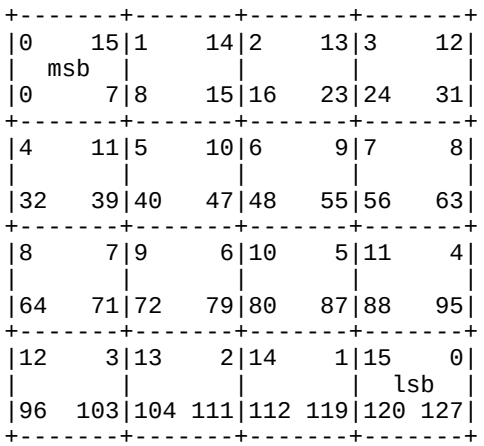


Figure 3-4. Bit and Byte Numbering in Quadwords

Fundamental Types

The following table shows how ANSI C scalar types correspond to those of the 64-bit PowerPC processor. For all types, a NULL pointer has the value zero. The alignment column specifies the required alignment of a field of the given type within a struct. The required alignment of a double or long double field is word, not doubleword or quadword as might be expected from the size of the field. Variables may be more strictly aligned than is shown in the table above, but fields in a struct must follow the alignment specified above in order to ensure consistent struct mapping.

"Extended precision" IBM AIX® 128-bit long double format composed of two double-precision numbers with different magnitudes that do not overlap. The high-order double-precision value (the one that comes first in storage) must have the larger magnitude. The value of the extended-precision number is the sum of the two double-precision values. For a value of NaN or infinity, you must encode one of these values within the high-order double-precision value. The low-order value is not significant. Extended precision provides the same range of double precision (about $10^{(-308)}$ to 10^{308}) but more precision (a variable amount, about 31 decimal digits or more). The software support is restricted to round-to-nearest mode. Programs that use extended precision must ensure that this rounding mode is in effect when extended-precision calculations are performed.

Type	ANSI C	sizeof	Alignment	PowerPC
Character	char unsigned char	1	byte	unsigned byte
	signed char	1	byte	signed byte
	short signed short	2	halfword	signed halfword
	unsigned short	2	halfword	unsigned halfword
Integral	int signed int enum	4	word	signed word
	unsigned int	4	word	unsigned word
	long int signed long long long	8	doubleword	signed doubleword
	unsigned long unsigned long long	8	doubleword	unsigned doubleword
Pointer	any * any (*) ()	8	doubleword	unsigned doubleword
Floating	float	4	word	single precision
	double	8	word	double precision

```
             ------------------------------------------------------
----
             long double     16        word         extended precision
```

Note: When compared to the 32-bit PowerPC Processor Supplement, the size and alignment of long has changed, and the alignment of double and long double has changed. A compiler may provide options to use different sizes and alignments; however, any object compiled with those options will not conform to the 64-bit PowerPC Processor Supplement.

Aggregates and Union

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, 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
+-------+
|0     0|
|   c   |
+-------+
```

Figure 3-5. Structure Smaller Than a Word

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

word aligned, sizeof is 8
little endian:

+-------+-------+-------+-------+
|              2|      1|      0|
|       s       |   d   |   c   |
+-------+-------+-------+-------+
```

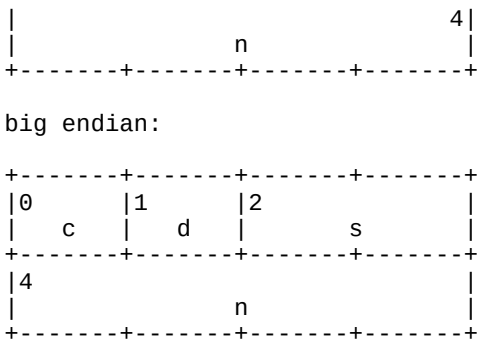


Figure 3-6. No Padding

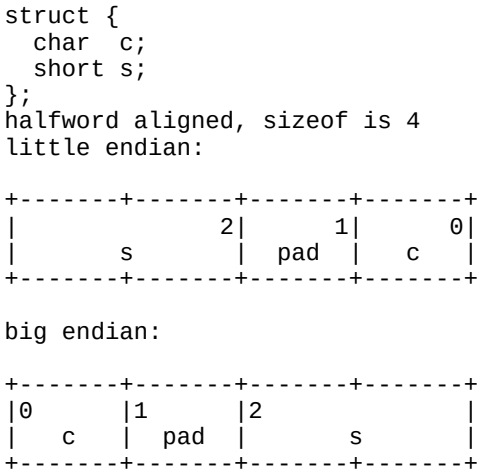
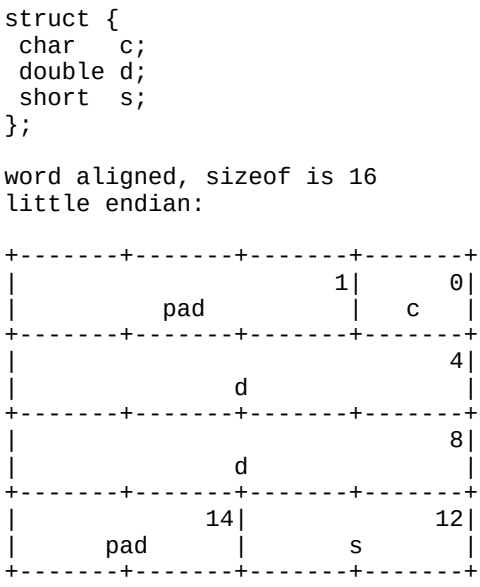


Figure 3-7. Internal Padding



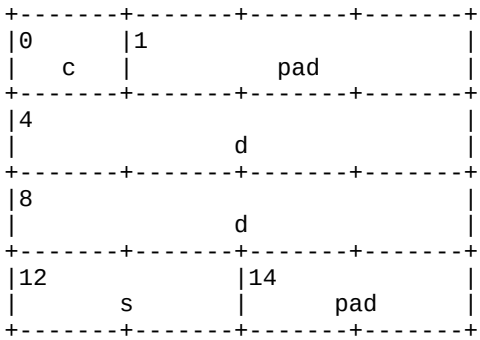
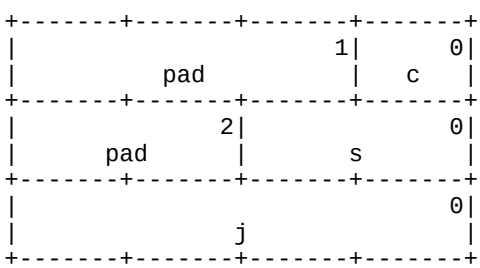


Figure 3-8. Internal and Tail Padding

```
union {
  char  c;
  short s;
  int   j;
};
word aligned, sizeof is 4
little endian:
```




```
big endian:
```

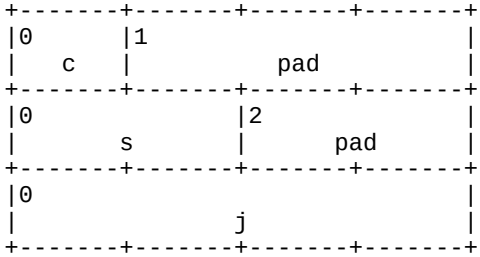


Figure 3-9. Union Allocation

Bit-fields

C struct and union definitions may have "bit-fields," defining integral objects with a specified number of bits.

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

*****Is it ok that I replaced the "**" with superscripts?*****

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

"Plain" bit-fields (that is, those neither signed nor unsigned) may have either positive or negative values, except in the case of plain char, which is always positive. 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.
- Bit-fields are limited to at most 64 bits. Adjacent bit-fields that cross a 64-bit boundary will start a new storage unit.
- The alignment of a bit-field is the same as the alignment of the base type of the bit-field. Thus, an int bit-field will have word alignment.
- 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.

> **Note:** The 64-bit PowerOpen ABI restricts bit-fields to be of type signed int, unsigned int, plain int, long, or unsigned long. This document does not have that restriction.
>
> The 32-bit PowerPC Processor Supplement specifies that a bit-field must entirely reside in a storage unit appropriate for its declared type. This document only restricts bit-fields to a 64-bit storage unit.

The following examples 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.

```
0x01020304

+-------+-------+-------+-------+
|0     3|1     2|2     1|3     0|
|  01   |  02   |  03   |  04   |
|0     7|8    15|16   23|24   31|
```

```
+-------+-------+-------+-------+
```

Figure 3-10. Bit Numbering

```
struct {
  int j : 5;
  int k : 6;
  int m : 7;
};
word aligned, sizeof is 4
little endian:

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

big endian:

+-----+------+-------+----------+
|0    |      |       |          |
|  j  |  k   |   m   |   pad    |
|0   4|5   10|11   17|18      31|
+-----+------+-------+----------+
```

Figure 3-11. Bit-field Allocation

```
struct {
  short s : 9;
  int   j : 9;
  char  c;
  short t : 9;
  short u : 9;
  char  d;
};
word aligned, sizeof is 8
little endian:

+-------+-----+--------+--------+
|      3|     |        |       0|
|   c   | pad |   j    |   s    |
|0     7|8  13|14    22|23    31|
+-------+-----+--------+--------+
|      7|     |        |       4|
|   d   | pad |   u    |   t    |
|0     7|8  13|14    22|23    31|
+-------+-----+--------+--------+

big endian:

+--------+--------+-----+-------+
|0       |        |     |3      |
|   s    |   j    | pad |   c   |
|0      8|9     17|18 23|24   31|
+--------+--------+-----+-------+
|4       |        |     |7      |
|   t    |   u    | pad |   d   |
|0      8|9     17|18 23|24   31|
+--------+--------+-----+-------+
```

Figure 3-12. Boundary Alignment

```
struct {
  long i : 56;
  int  j : 9:
};
doubleword aligned, sizeof is 16
little endian:

+--------------------------------+
|                                0|
|              i                 |
|0                              31|
+-------+------------------------+
|       |                        4|
|  pad  |         i              |
|32   39|40                     63|
+-------+--------------+---------+
|                      |        8|
|         pad          |   j     |
|0                    22|23    31|
+----------------------+---------+
|                               12|
|             pad                |
|0                              31|
+--------------------------------+

big endian:

+--------------------------------+
|0                               |
|              i                 |
|0                              31|
+-----------------------+-------+
|4                      |       |
|           i           |  pad  |
|32                    55|56   63|
+--------+--------------+-------+
|8       |                       |
|   j    |        pad            |
|0      8|9                     31|
+----------------------+---------+
|12                              |
|             pad                |
|0                              31|
+--------------------------------+
```

Figure 3-13. Doubleword Boundary Alignment

```
struct {
  char  c;
  short s : 8;
};
halfword aligned, sizeof is 2
little endian:

+-------+-------+
|      1|      0|
|   s   |   c   |
|0     7|8    15|
+-------+-------+

big endian:

+-------+-------+
|0      |1      |
|   c   |   s   |
```

```
|0     7|8    15|
+-------+-------+
```

Figure 3-14. Storage Unit Sharing

```
union {
  char  c;
  short s : 8;
};
halfword aligned, sizeof is 2
little endian:

+-------+-------+
|      1|      0|
|  pad  |   c   |
|0     7|8    15|
+-------+-------+
|      1|      0|
|  pad  |   s   |
|0     7|8    15|
+-------+-------+

big endian:

+-------+-------+
|0      |1      |
|   c   |  pad  |
|0     7|8    15|
+-------+-------+
|0      |1      |
|   s   |  pad  |
|0     7|8    15|
+-------+-------+
```

Figure 3-15. Union Allocation

```
struct {
  char  c;
  int   : 0;
  char  d;
  short : 9;
  char  e;
};
byte aligned, sizeof is 8
little endian:

+-----------------------+-------+
|                      1|      0|
|           :0          |   c   |
|0                    23|24   31|
+-------+------+--------+-------+
|      7|      |        |      4|
|   e   | pad  |   :9   |   d   |
|0     7|8   14|15    23|24   31|
+-------+------+--------+-------+

big endian:

+-------+-----------------------+
|0      |1                      |
|   c   |          :0           |
|0     7|8                    31|
+-------+--------+------+-------+
```

```
|4      |        |      |7      |
|   d   |   :9   | pad  |   e   |
|0     7|8     16|17  23|24   31|
+-------+--------+------+-------+
```

Figure 3-16. Unnamed bit-fields

> **Note:** In this example, the presence of the unnamed int and short fields does 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.

Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, and parameter passing.

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

> **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 provide traceback tables as described in the Section called *Traceback Tables*. Nonetheless, it is recommended that all functions use the standard calling sequences when possible.

Registers

The 64-bit PowerPC Architecture provides 32 general purpose registers, each 64 bits wide. In addition, the architecture provides 32 floating-point registers, each 64 bits wide, and several special purpose registers. All of the integer, special purpose, and floating-point registers are global to all functions in a running program. The following table shows how the registers are used.

```
r0        Volatile register used in function prologs
r1        Stack frame pointer
r2        TOC pointer
r3        Volatile parameter and return value register
r4-r10    Volatile registers used for function parameters
r11       Volatile register used in calls by pointer and as an
          environment pointer for languages which require one
r12       Volatile register used for exception handling and glink code
r13       Reserved for use as system thread ID
r14-r31   Nonvolatile registers used for local variables

f0        Volatile scratch register
f1-f4     Volatile floating point parameter and return value registers
f5-f13    Volatile floating point parameter registers
f14-f31   Nonvolatile registers

LR        Link register (volatile)
CTR       Loop counter register (volatile)
XER       Fixed point exception register (volatile)
FPSCR     Floating point status and control register (volatile)

CR0-CR1   Volatile condition code register fields
```

```
CR2-CR4   Nonvolatile condition code register fields
CR5-CR7   Volatile condition code register fields
```

Registers r1, r14 through r31, and f14 through f31 are nonvolatile, which means that they preserve their values across function calls. Functions which use those registers must save the value before changing it, restoring it before the function returns. Register r2 is technically nonvolatile, but it is handled specially during function calls as described below: in some cases the calling function must restore its value after a function call.

Registers r0, r3 through r12, f0 through f13, and the special purpose registers LR, CTR, XER, and FPSCR are volatile, which means that they are not preserved across function calls. Furthermore, registers r0, r2, r11, and r12 may be modified by cross-module calls, so a function can not assume that the values of one of these registers is that placed there by the calling function.

The condition code register fields CR0, CR1, CR5, CR6, and CR7 are volatile. The condition code register fields CR2, CR3, and CR4 are nonvolatile; a function which modifies them must save and restore at least those fields of the CR. Languages that require "environment pointers" shall use r11 for that purpose.

The following registers have assigned roles in the standard calling sequence:

r1

The stack pointer (stored in r1) shall maintain quadword alignment. 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 the Section called *Dynamic Stack Space Allocation* for additional infromation. As discussed later in this chapter, the lowest valid stack address is 288 bytes less than the value in the stack pointer. The stack pointer must be atomically updated by a single instruction, thus avoiding any timing window in which an interrupt can occur with a partially updated stack.

r2

This register holds the TOC base. See the Section called *The TOC section* for additional information.

r3 through r10 and f1 through f13

r3 through r10 and f1 through f13: 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 the Section called *Parameter Passing* for additional information. In addition, registers r3 and f1 through f4 are used to return values from the called function, as described in Return Values.

LR (Link Register)

This 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.

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. The following figure 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.

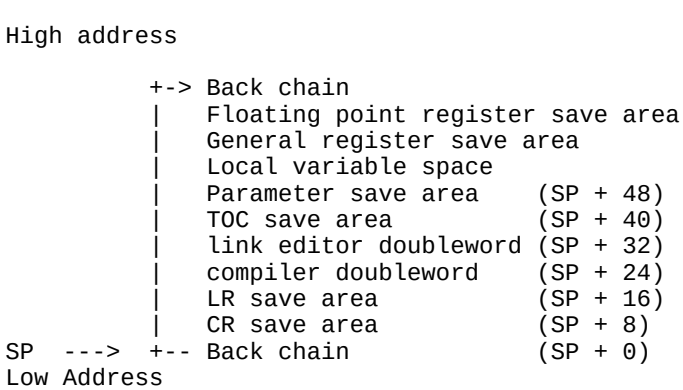


Figure 3-17. Stack Frame Organiztion

*****Is the low address example missing?*****

The following requirements apply to the stack frame:

- The stack pointer shall maintain quadword 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 sizes of the floating-point and general register save areas may vary within a function and are as determined by the traceback table described below.
- Before a function changes the value in any nonvolatile floating-point register, frn, it shall save the value in frn in the double word in the floating-point register save area 8*(32-n) bytes before the back chain word of the previous frame. The floating-point register save area is always doubleword aligned. The size of the floating-point register save area depends upon the number of floating point registers which must be saved. It ranges from 0 bytes to a maximum of 144 bytes (18 * 8).
- Before a function changes the value in any nonvolatile general register, rn, it shall save the value in rn in the word in the general register save area 8*(32-n) bytes before the low addressed end of the floating-point register save area. The general register save area is always doubleword aligned. The size of the general register save area depends upon the number of general registers which must be saved. It ranges from 0 bytes to a maximum of 144 bytes (18 * 8).
- The local variable space contains any local variable storage required by the function.
- The parameter save area shall be allocated by the caller. It shall be doubleword aligned, and shall be at least 8 doublewords in length. If a function needs to pass more than 8 doublewords of arguments, the parameter save area shall be large enough to contain the arguments that the caller stores in it. Its contents are not preserved across function calls.

- The TOC save area is used by global linkage code to save the TOC pointer register. See The TOC section later in the chapter.
- The link editor doubleword is reserved for use by code generated by the link editor. This ABI does not specify any usage; the AIX link editor uses this space under certain circumstances.
- The compiler doubleword is reserved for use by the compiler. This ABI does not specify any usage; the AIX compiler uses this space under certain circumstances.
- Before a function calls any other functions, it shall save the value in the LR register in the LR save area.
- 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.
- A function which does not call any other function and which needs no local variables does not need to have a stack frame. It may use non-volatile registers, saving their old values on the stack. Thus, the lowest possible valid stack address is 288 bytes (144 + 144) less than the current value of the stack pointer. Functions may use this stack space as volatile storage which is not preserved across function calls. Interrupt handlers and any other code which can not be seen by compiled code must avoid using this area.

The stack frame header consists of the back chain word, the CR save area, the LR save area, the compiler and link editor doublewords, and the TOC save area, for a total of 48 bytes. 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 area 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 save area shall immediately follow the stack frame header, and the register save areas shall contain no padding.

Parameter Passing

For a RISC machine such as 64-bit PowerPC, it is generally more efficient to pass arguments to called functions in registers (both general and floating-point 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 the 64-bit PowerPC, up to eight doublewords are passed in general purpose registers, loaded sequentially into general purpose registers r3 through r10. In addition, up to thirteen floating-point arguments can be passed in floating-point registers f1 through f13. If fewer (or no) arguments are passed, the unneeded registers are not loaded and will contain undefined values on entry to the called function.

The parameter save area, which is located at a fixed offset of 48 bytes from the stack pointer, is reserved in each stack frame for use as an argument list. A minimum of 8 doublewords is always reserved. The size of this area must be sufficient to hold the longest argument list being passed by the function which owns the stack frame. Although not all arguments for a particular call are located in storage, consider them

to be forming a list in this area, with each argument occupying one or more doublewords.

If more arguments are passed than can be stored in registers, the remaining arguments are stored in the parameter save area. The values passed on the stack are identical to those that have been placed in registers; thus, the stack contains register images.

The rules for parameter passing are as follows:

- Each argument is mapped to as many doublewords of the parameter save area as are required to hold its value.
  - Single precision floating point values are mapped to the first word in a single doubleword.
  - Double precision floating point values are mapped to a single doubleword.
  - Extended precision floating point values are mapped to two consecutive doublewords.
  - Simple integer types (char, short, int, long, enum) are mapped to a single doubleword. Value shorter than a doubleword are sign or zero extended as necessary.
  - Pointers are mapped to a single doubleword.
  - Aggregates and unions passed by value are mapped to as many words of the parameter save area as the value uses in memory.
  - Other scalar values, such as FORTRAN complex numbers, are mapped to the number of doublewords required by their size.

- If the callee has a known prototype, arguments are converted to the type of the corresponding parameter before being mapped into the parameter save area. For example, if a long is used as an argument to a float double parameter, the value is converted to double-precision and mapped to a doubleword in the parameter save area.
- The first eight doublewords mapped to the parameter save area are never stored in the parameter save area by the calling function. Instead, these doublewords are passed in registers as described below.
- Argument values beyond the first eight doublewords must be stored in the parameter save area following the first eight doublewords, even though these arguments may also be passed in the floating point registers as described below. The first eight doublewords in the parameter save area are reserved for the initial arguments, even though they are passed in registers.
- General registers are used to pass some values. The first eight doublewords mapped to the parameter save area correspond to the register r3 through r10. If the arguments are mapped to fewer than eight doublewords of the parameter save area, registers corresponding to those unused doublewords are not used.
  - If there is no known function prototype for a callee, or the known prototype contains an ellipsis, the first eight doublewords mapped to the parameter save area are passed in their corresponding general registers.
  - If the callee has a prototype with no ellipsis, values other than single and double precision floating point values are passed in their corresponding general registers. General registers corresponding to single and double precision floating-point arguments are not used--the floating point values are placed in floating-point registers instead.

- Floating point registers are used consecutively to pass up to 13 single and double precision floating point values. Each floating point register holds at most a single value, whether the argument is single or double precision.
  - If there is no known prototype for a callee, or the known prototype contains an ellipsis, up to 13 floating point values are passed in f1 through f13. In this case, floating point values mapped to the first eight doublewords of the parameter save area are also passed in general registers.
  - If there is a known prototype for the callee, and it doesn't contain an ellipsis, up to 13 floating point values are passed in f1 through f13. In this case, floating point values mapped to the first eight doublewords of the parameter save argument area are *not* also passed in the general registers.

- If the callee takes the address of any of its parameters, then r3 through r10 must be stored by the callee into the parameter save area, except when any of these general registers correspond to single or double precision floating point parameters, in which case the floating point registers are stored rather than the general registers. This is because the caller may have had a function prototype for the callee, and therefore put floating point parameters in floating point registers only.

If the compilation unit for the caller contains a function prototype, but the callee has a mismatching definition, and if the callee takes the address of any of its parameters, the wrong values may be stored in the first eight doublewords of the parameter save area.

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

x = func(c, ff, d, ld, s, gg, t, e, hh);
Parameter     Register     Offset in parameter save area
c             r3           0-7    (not stored in parameter save area)
ff            f1           8-15   (not stored)
d             r5           16-23  (not stored)
ld            r6,r7        24-39  (not stored)
s             r8,r9        40-55  (not stored)
gg            f2           56-63  (not stored)
t             (none)       64-79  (stored in parameter save area)
e             (none)       80-87  (stored)
hh            f3           88-95  (stored)
```

Figure 3-18. Parameter Passing

The above is complete if a prototype with no ellipsis is in scope. If a prototype is not in scope, or if the prototype contains an ellipsis, then, in addition to the above, ff is passed in r4 and gg is passed in r10. The floating point argument hh is passed in two places: the value is stored both in f3 and on the stack. If a prototype is not in scope, or if the prototype contains an ellipsis, then the floating point arguments ff and gg are also passed in two places, in both a general register and a floating point register.

Return Values

Functions shall return float or double values in f1, with float values rounded to single precision.

Functions shall return values of type int, long, enum, short, and char, or a pointer to any type, as unsigned or signed integers as appropriate, zero- or sign-extended to 64 bits if necessary, in r3. Character arrays of length 8 bytes or less, or bit strings of length 64 bits or less, will be returned right justified in r3. Aggregates or unions of any length, and character strings of length longer than 8 bytes, will be returned in a storage buffer allocated by the caller. The caller will pass the address of this buffer as a hidden first argument in r3, causing the first explicit argument to be passed in r4. This hidden argument is treated as a normal formal parameter, and corresponds to the first doubleword of the parameter save area.

Functions shall return floating point scalar values of size 16 or 32 bytes in f1:f2 and f1:f4, respectively.

Functions shall return floating point complex values of size 16 (four or eight byte complex) in f1:f2 and floating point complex values of size 32 (16 byte complex) in f1:f4.

Function Descriptors

A function descriptor is a three doubleword data structure that contains the following values:

- The first doubleword contains the address of the entry point of the function.
- The second doubleword contains the TOC base address for the function (see The TOC Section later in this chapter).
- The third doubleword contains the environment pointer for languages such as Pascal and PL/1.

For an externally visible function, the value of the symbol with the same name as the function is the address of the function descriptor. Symbol names with a dot (.) prefix are reserved for holding entry point addresses. The value of a symbol named ".FN" is the entry point of the function FN".

The value of a function pointer in a language like C is the address of the function descriptor. Examples of calling a function through a pointer are provided below.

*****Where are the examples?"*****

When the link editor processes relocatable object files in order to produce an executable or shared object, it must treat direct function calls specially, as described below.

Traceback Tables

To support debuggers and exception handlers, the 64-bit PowerPC ELF ABI defines traceback tables. Compilers must support generation of at least the mandatory part of traceback tables, and system libraries should contain the mandatory part. Compilers should provide an option to turn off traceback table generation to save space when the information is not needed.

Traceback tables are intended to be compatible with the 64-bit PowerOpen ABI.

Compilers should generate a traceback table following the end of the code for every function. Debuggers and exception handlers can locate the traceback tables by scanning forward from the instruction address at the point of interruption. The beginning of the traceback table is marked by a word of zeroes, which is an illegal instruction. If read-only constants are compiled into the same section as the function code, they must follow the traceback table. A word of zeroes as read-only data must not be the first word following the code for a function. A traceback table is word-aligned.

Mandatory Fields

The following are the mandatory fields of a traceback table:

version Eight-bit field. This defines the type code for the table. The only currently defined value is zero.

lang Eight-bit field. This defines the source language for the compiler that generated the code for which this traceback table applies. The default values are as follows:

C	0
FORTRAN	1
Pascal	2
Ada	3
PL/1	4
Basic	5
LISP	6
COBOL	7
Modula2	8
C++	9
RPG	10
PL.8,PLIX	11
Assembly	12

The codes 0xd to 0xfa are reserved. The codes 0xfb to 0xff are reserved for IBM.

globalink One-bit field. This field is set to 1 if this routine is a special routine used to support the linkage convention: a linkage function or a ._ptrgl function. See the section Function Calls for more information. These routines have unusual register usage and stack format.

is_eprol One-bit field. This field is set to 1 if this routine is an out-of-line prologue or epilogue function. See the section Function Prologue and Epilogue for more information. These routines have unusual register usage and stack format.

has_tboff One-bit field. This field is set to 1 if the offset of the traceback table from the start of the function is stored in the tb_offset field.

int_proc One-bit field. This field is set to 1 if this function is a stackless leaf function that does not have a separate stack frame.

has_ctl One-bit field. This field is set to 1 if ctl_info is provided.

tocless One-bit field. This field is set to 1 if this function does not have a TOC. For example, a stackless leaf assembly language routine with no references to external objects.

fp_present One-bit field. This field is set to 1 if the function uses floating-point processor instructions.

log_abort One-bit field. Reserved.

int_handl One-bit field. Reserved.

name_present One-bit field. This field is set to 1 if the name for the procedure is present following the traceback field, as determined by the name_len and name fields.

uses_alloca	One-bit field. This field is set to 1 if the procedure performs dynamic stack allocation. To address their local variables, these procedures require a different register to hold the stack pointer value. This register may be chosen by the compiler, and must be indicated by setting the value of the alloc_reg field.
cl_dis_inv	Three-bit field. Reserved.
saves_cr	One-bit field. This field is set to 1 if the function saves the CR in the CR save area.
saves_lr	One-bit field. This field is set to 1 if the function saves the LR in the LR save area.
stores_bc	One-bit field. This field is set to 1 if the function saves the back chain (the SP of its caller) in the stack frame header.
fixup	One-bit field. This field is set to 1 if the link editor replaced the original instruction by a branch instruction to a special fixup instruction sequence.
fp_saved	Six-bit field. This field is set to the number of non-volatile floating point registers that the function saves. The last register saved is always f31, so, for example, a value of 2 in this field indicates that f30 and f31 are saved.
spare3	Two-bit field. Reserved.
gpr_saved	Six-bit field. This field is set to the number of non-volatile general registers that the function saves. As with fp_saved, the last register saved is always r31.
fixedparms	Eight-bit field. This field is set to the number of fixed point parameters.
floatparms	Seven-bit field. This field is set to the number of floating point parameters.
parmsonstk	One-bit field. This field is set to 1 if all of the parameters are placed in the parameter save area.

Note: If either fixedparms or floatparms is set to a non-zero value, the parminfo field exists.

A debugger can use the fixedparms, floatparms, and parmsonstk field to support displaying the parameters passed to a function. They specify the number of parameters passed in the general registers and the number passed in the floating point registers; they also specify whether the parameters are stored in the parameter save area. The parameters are stored in the parameter save area if the number of parameters is variable, or if the address of one of the parameters is taken, or if the compiler always stores the parameters at the optimization level of the compilation. If either the fixedparms or floatparms field is set to a non-zero value, then the next field, parminfo, can be used by a debugger to determine the relative order and types of the parameters.

Optional Fields

The following are the optional fields of a traceback table:

parminfo — Unsigned int. This field is only present if either fixedparms or floatparms is set to a non-zero value. It can be used by a debugger to determine which registers were used to pass parameters to the routine and to determine the layout of the parameter save area. This word is interpreted from left to right, as follows:

- bit is 0: the corresponding parameter is a fixed point parameter passed in a general register or a single doubleword in the parameter save area.
- bit is 1: the corresponding parameter is a floating point parameter, and the following bit determines whether the parameter is single precision (the following bit is 0) or double precision (the following bit is 1).

Note: Since this field is only 32 bits long, there is a limit to how many parameters can be described. This limit is in the range of 16 to 32 parameters depending upon the type of the parameters. Note that it takes two bits to describe a floating point parameter and one bit for each non floating point parameter.

tb_offset — Unsigned int. This word is only present if the has_tboff field is set to 1. It holds the length of the function code.

hand_mask — Int. Reserved.

ctl_info — Int. This word is only present if the has_ctl field is set to 1. It gives the number of controlled automatic anchor blocks defined for this procedure. If an exception handler is unwinding the stack to restart some earlier function, the the controlled automatic storage must be released. Controlled automatic storage is used by PL/1 and PL.8.

ctl_info_disp — Int[*]. This field is only present if the has_ctl field is set to 1. The ctl_info field indicates the number of words. Each word is the displacement to the location of the information.

name_len — Short. This field is only present if the name_present field is set to 1. It is the length of the function name that immediately follows this field.

name — char[*]. This field is only present if the name_present field is set to 1. The name_len field indicates the number of characters. The name is in seven-bit ASCII, and is not delimited by a null character.

alloca_reg — Char. This field is only present if the uses_alloca bit is set to 1. It holds the register number that is used as the base for variable accesses.

Process Initialization

This section describes the machine state that exec creates for "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 as follows:

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

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.

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 set all registers to 0. Following are the registers whose contents are specified:

r1

The initial stack pointer, aligned to a quadword boundary and pointing to a word containing a NULL pointer.

r2

The initial TOC pointer register value, obtained via the function descriptor pointed at by the e_entry field in the ELF header. For more information on function decscriptors, see the Section called *Function Descriptors*. For more information on the ELF Header, see Chapter 4.

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 below).

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.

fpscr

Contains 0, specifying "round to nearest" mode, IEEE Mode, and the disabling of floating-point exceptions.

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, defined as follows:

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

Name	Value	a_un field
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

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 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 Program Loading in Chapter 5 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 the sysconf system call.

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 64-bit PowerPC Architecture.

When a process receives control, its stack holds the arguments, environment, and auxiliary vector from exec. 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).

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.

64-bit PowerPC code is normally position independent. That is, the code is not tied to a specific load address, and may be executed properly at various positions in virtual memory. Although it is possible to write position dependent code on the 64-bit PowerPC, these code examples only show position independent code.

> **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.

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 the 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 64-bit 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," or 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.

The TOC section

ELF processor-specific supplements normally define a GOT ("Global Offset Table") section used to hold addresses for position independent code. Some ELF processor-specific supplements, including the 32-bit PowerPC Processor Supplement, define a small data section. The same register is sometimes used to address both the GOT and the small data section.

The 64-bit PowerOpen ABI defines a TOC ("Table of Contents") section. The TOC combines the functions of the GOT and the small data section.

This ABI uses the term TOC. The TOC section defined here is intended to be similar to that defined by the 64-bit PowerOpen ABI. The TOC section contains a conventional ELF GOT, and may optionally contain a small data area. The GOT and the small data area may be intermingled in the TOC section.

The TOC section is accessed via the dedicated TOC pointer register, r2. Accesses are normally made using the register indirect with immediate index mode supported by the 64-bit PowerPC processor, which limits a single TOC section to 65,536 bytes, enough for 8,192 GOT entries.

The value of the TOC pointer register is called the TOC base. The TOC base is typically the first address in the TOC plus 0x8000, thus permitting a full 64 Kbyte TOC.

A relocatable object file must have a single TOC section and a single TOC base. However, when the link editor combines relocatable object files to form a single executable or shared object, it may create multiple TOC sections. The link editor is responsible for deciding how to associate TOC sections with object files. Normally the link editor will only create multiple TOC sections if it has more than 65,536 bytes to store in a TOC.

All link editors which support this ABI must support a single TOC section, but support for multiple TOC sections is optional.

Each shared object will have a separate TOC or TOCs.

> **Note:** This ABI does not actually restrict the size of a TOC section. It is permissible to use a larger TOC section, if code uses a different addressing mode to access it. The AIX link editor, in particular, does not support multiple TOC sections, but instead inserts call out code at link time to support larger TOC sections.

TOC Assembly Language Syntax

Desire for compatibility with both ELF systems and PowerOpen systems suggests two different assembly language syntaxes to be used when referring to the TOC section. This syntax is not part of the official ABI. The description here is only for information purposes. Particular assemblers may support both syntaxes, only one, or neither.

The ELF syntax uses @got and @toc. The syntax SYMBOL@got refers to the offset in the TOC at which the value of SYMBOL (that is, the address of the variable whose name is SYMBOL) is stored, assuming the offset is no larger than 16 bits. For example,

```
ld   r3,x@got(r2)
```

SYMBOL@got will be an offset within the global offset table, which as noted above, forms part of the TOC section.

Ordinarily the link editor will avoid having a TOC, and hence a GOT, larger than 64 Kbytes, perhaps by support multiple TOC sections, or via some other technique. However, for flexibility, there is a syntax for 32 bit offsets to the GOT. The syntaxes SYMBOL@got@ha, SYMBOL@got@h, and SYMBOL@got@l refer to the high

adjusted, high, and low parts of the GOT offset. (The meaning of "high adjusted" is explained in the Section called *Relocation Types* in Chapter 4).

The syntax SYMBOL@toc refers to the value (SYMBOL - base (TOC)), where base (TOC) represents the TOC base for the current object file. This provides the address of the variable whose name is SYMBOL, as an offset from the TOC base. This assumes that the variable may be found within the TOC, and that its offset is no larger than 16 bits.

As with the GOT, the syntaxes SYMBOL@toc@ha, SYMBOL@toc@h, and SYMBOL@toc@l refer to the high adjusted, high, and low parts of the TOC offset.

The syntax SYMBOL@got@plt may be used to refer to the offset in the TOC of a procedure linkage table entry stored in the global offset table. The corresponding syntaxes SYMBOL@got@plt@ha, SYMBOL@got@plt@h, and SYMBOL@got@plt@l are also defined.

> **Note:** Note that if X is a variable stored in the TOC, then X@got will be the offset within the TOC of a doubleword whose value is X@toc.

The special symbol .TOC.@tocbase is used to represent the TOC base for the current object file. The following might appear in a function descriptor definition:

```
.quad .TOC.@tocbase
```

The PowerOpen syntax is more complex. It is derived from the different representation of the TOC section in XCOFF.

Assembly code first uses the .toc pseudo-op to enter the TOC section. It then uses a label to name a particular element. It then uses the .tc pseudo-op to indicate which GOT entry it wishes to name. Later in the code, the label is used with the TOC register to load the address. For example:

```
    .toc
.L1:
    .tc  x[TC],x
    ...
    ld   r3,.L1(r2)
```

This creates a GOT entry for the variable x, and names that entry .L1 for the remainder of the assembly. The effect is the same as the single ELF-style instruction above.

The special value TOC[tc0] is used to represent the TOC base for the current object file:

```
.quad TOC[tc0]
```

The PowerOpen syntax permits other data to be stored in the .toc section. The assembler will output this data in a .toc section, and convert references as though its address were specified with @toc rather than @got.

There is a significant difference in representation of the TOC in this ABI and in the 64-bit PowerOpen ABI. Relocatable object files created using the 64-bit PowerOpen ABI have a .toc section which contains real data. The link editor uses garbage collection to discard duplicate information including in particular TOC entries which refer to the same variable. In this ABI, relocatable object files do not contain .got sections holding real data. Instead, the GOT is created by the link editor based on relocations created by @got references. This ABI does not require the link editor to support garbage collection. This ABI does permit real data to exist in .toc sections, but this data will never be referred to directly by instructions which use @got references. @got references always refer to the GOT which is created by the link editor when creating an executable or a shared object.

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:

- If the function uses any nonvolatile general registers, it shall save them in the general register save area. If the function does not require a stack frame, this may be done using negative stack offsets from the caller's stack pointer.
- If the function uses any nonvolatile floating point registers, it shall save them in the floating point register save area. If the function does not require a stack frame, this may be done using negative stack offsets from the caller's stack pointer.
- Before a function calls any other function, it shall establish its own stack frame, whose size shall be a multiple of 16 bytes, and shall save the link register at the time of entry in the LR save area of its caller's stack frame.
- If the function uses any nonvolatile fields in the CR, it shall save the CR in the CR save area of the caller's stack frame.
- 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 Double Word with Update" instructions.
  - For small (no larger than 32 Kbytes) stack frames, this may be accomplished with a "Store Double 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 Double Word with Update Indexed" instruction.
- 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 or floating-point 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.

Register Saving and Restoring Functions

The register saving and restoring functions described in this section use nonstandard calling conventions which ordinarily require them to be statically linked into any executable or shared object modules in which they are used. Nevertheless, unlike 32-bit PowerPC ELF, these functions are considered part of the official ABI. In particular, the link editor is permitted to treat calls to these functions specially, such as by changing a call to one of these function into a call to an absolute address as in the PowerOpen ABI.

As shown in The Stack Frame section above, the general register save area is not at a fixed offset from either the caller's SP or the callee's SP. The floating point register save area starts at a fixed position from the caller's SP on entry to the callee, but the position of the general register save area depends upon the number of floating point registers to be saved. Thus it is impossible to write a general register saving routine which uses fixed offsets from the SP.

If the routine needs to save both general and floating point registers, code can use r12 as the pointer for saving and restoring the general purpose registers. (r12 is a volatile register but does not contain input parameters). This leads to the definition of multiple register save and restore routines, each of which saves or restores M floating point registers and N general registers.

Saving General Registers Only

For a function that saves/restores N general registers and no floating point registers, the saving can be done using individual store/load instructions or by calling system provided routines as shown below.

In the following, the number of registers being saved is N, and <32-N> is the first register number to be saved/restored. All registers from <32-N> up to 31, inclusive, are saved/restored.

FRAME_SIZE is the size of the stack frame, here assumed to be less than 32 Kbytes.

```
    mflr  r0                    # Move LR into r0
    bl    _savegpr0_<32-N>       # Call routine to save general registers
    stdu  r1,(-FRAME_SIZE)(r1)  # Create stack frame
    ...
    (save CR if necessary)
    ...                         # Body of function
    ...
    (reload CR if necessary)
    ...
    (reload caller's SP into r1)
    b     _restgpr0_<32-N>       # Restore registers and return
```

Saving General Registers and Floating Point Registers

For a function that saves/restores N general registers and M floating point registers, the saving can be done using individual store/load instructions or by calling system provided routines as shown below.

```
    mflr  r0                    # Move LR into r0
    subi  r12,r1,8*M            # Set r12 to general reg save area
    bl    _savegpr1_<32-N>       # Call routine to save general registers
    bl    _savefpr_<32-M>        # Call routine to save floating point regs
    stdu  r1,(-FRAME_SIZE)(r1)  # Create stack frame
    ...
    (save CR if necessary)
    ...                         # Body of function
    ...
    (reload CR if necessary)
    ...
    (reload caller's SP into r1)
    subi  r12,r1,8*M            # Set r12 to general reg save area
    bl    _restgpr1_<32-N>       # Restore general registers
    b     _restfpr_<32-M>        # Restore floating point regs and re-
turn
```

Saving Floating Point Registers Only

For a function that saves/restores M floating point registers and no general registers, the saving can be done using individual store/load instructions or by calling system provided routines as shown below.

```
    mflr  r0                    # Move LR into r0
```

```
bl    _savefpr_<32-M>       # Call routine to save general registers
stdu  r1,(-FRAME_SIZE)(r1)  # Create stack frame
...
(save CR if necessary)
...                         # Body of function
...
(reload CR if necessary)
...
(reload caller's SP into r1)
b     _restgpr_<32-M>       # Restore registers and return
```

Save and Restore Services

Systems must provide three sets of routines, which may be implemented as multiple entry point routines or as individual routines. They must adhere to the following rules.

Each _savegpr0_N routine saves the general registers from rN to r31, inclusive. Each routine also saves the LR. When the routine is called, r1 must point to the start of the general register save area, and r0 must contain the value of LR on function entry.

The _restgpr0_N routines restore the general registers from rN to r31, and then return to the caller. When the routine is called, r1 must point to the start of the general register save area.

Here is a sample implementation of _savegpr0_N and _restgpr0_N.

```
_savegpr0_14:  std  r14,-144(r1)
_savegpr0_15:  std  r15,-136(r1)
_savegpr0_16:  std  r16,-128(r1)
_savegpr0_17:  std  r17,-120(r1)
_savegpr0_18:  std  r18,-112(r1)
_savegpr0_19:  std  r19,-104(r1)
_savegpr0_20:  std  r20,-96(r1)
_savegpr0_21:  std  r21,-88(r1)
_savegpr0_22:  std  r22,-80(r1)
_savegpr0_23:  std  r23,-72(r1)
_savegpr0_24:  std  r24,-64(r1)
_savegpr0_25:  std  r25,-56(r1)
_savegpr0_26:  std  r26,-48(r1)
_savegpr0_27:  std  r27,-40(r1)
_savegpr0_28:  std  r28,-32(r1)
_savegpr0_29:  std  r29,-24(r1)
_savegpr0_30:  std  r30,-16(r1)
_savegpr0_31:  std  r31,-8(r1)
               std  r0, 16(r1)
               blr


_restgpr0_14:  ld   r14,-144(r1)
_restgpr0_15:  ld   r15,-136(r1)
_restgpr0_16:  ld   r16,-128(r1)
_restgpr0_17:  ld   r17,-120(r1)
_restgpr0_18:  ld   r18,-112(r1)
_restgpr0_19:  ld   r19,-104(r1)
_restgpr0_20:  ld   r20,-96(r1)
_restgpr0_21:  ld   r21,-88(r1)
_restgpr0_22:  ld   r22,-80(r1)
_restgpr0_23:  ld   r23,-72(r1)
_restgpr0_24:  ld   r24,-64(r1)
_restgpr0_25:  ld   r25,-56(r1)
_restgpr0_26:  ld   r26,-48(r1)
_restgpr0_27:  ld   r27,-40(r1)
_restgpr0_28:  ld   r28,-32(r1)
_restgpr0_29:  ld   r0, 16(r1)
```

```
                ld   r29,-24(r1)
                mtlr r0
                ld   r30,-16(r1)
                ld   r31,-8(r1)
                blr
_restgpr0_30:   ld   r30,-16(r1)
_restgpr0_31:   ld   r0, 16(r1)
                ld   r31,-8(r1)
                mtlr r0
                blr
```

Each _savegpr1_N routine saves the general registers from rN to r31, inclusive. When the routine is called, r12 must point to the start of the general register save area.

The _restgpr1_N routines restore the general registers from rN to r31. When the routine is called, r12 must point to the start of the general register save area.

Here is a sample implementation of _savegpr1_N and _restgpr1_N.

```
_savegpr1_14:  std  r14,-144(r12)
_savegpr1_15:  std  r15,-136(r12)
_savegpr1_16:  std  r16,-128(r12)
_savegpr1_17:  std  r17,-120(r12)
_savegpr1_18:  std  r18,-112(r12)
_savegpr1_19:  std  r19,-104(r12)
_savegpr1_20:  std  r20,-96(r12)
_savegpr1_21:  std  r21,-88(r12)
_savegpr1_22:  std  r22,-80(r12)
_savegpr1_23:  std  r23,-72(r12)
_savegpr1_24:  std  r24,-64(r12)
_savegpr1_25:  std  r25,-56(r12)
_savegpr1_26:  std  r26,-48(r12)
_savegpr1_27:  std  r27,-40(r12)
_savegpr1_28:  std  r28,-32(r12)
_savegpr1_29:  std  r29,-24(r12)
_savegpr1_30:  std  r30,-16(r12)
_savegpr1_31:  std  r31,-8(r12)
               blr


_restgpr1_14:  ld   r14,-144(r12)
_restgpr1_15:  ld   r15,-136(r12)
_restgpr1_16:  ld   r16,-128(r12)
_restgpr1_17:  ld   r17,-120(r12)
_restgpr1_18:  ld   r18,-112(r12)
_restgpr1_19:  ld   r19,-104(r12)
_restgpr1_20:  ld   r20,-96(r12)
_restgpr1_21:  ld   r21,-88(r12)
_restgpr1_22:  ld   r22,-80(r12)
_restgpr1_23:  ld   r23,-72(r12)
_restgpr1_24:  ld   r24,-64(r12)
_restgpr1_25:  ld   r25,-56(r12)
_restgpr1_26:  ld   r26,-48(r12)
_restgpr1_27:  ld   r27,-40(r12)
_restgpr1_28:  ld   r28,-32(r12)
_restgpr1_29:  ld   r29,-24(r12)
_restgpr1_30:  ld   r30,-16(r12)
_restgpr1_31:  ld   r31,-8(r12)
               blr
```

Each _savefpr_M routine saves the floating point registers from fM to f31, inclusive. When the routine is called, r1 must point to the start of the floating point register save area, and r0 must contain the value of LR on function entry.

The _restfpr_M routines restore the floating point registers from fM to f31. When the routine is called, r1 must point to the start of the floating point register save area.

Here is a sample implementation of _savepr_M and _restfpr_M.

```
_savefpr_14:  stfd f14,-144(r1)
_savefpr_15:  stfd f15,-136(r1)
_savefpr_16:  stfd f16,-128(r1)
_savefpr_17:  stfd f17,-120(r1)
_savefpr_18:  stfd f18,-112(r1)
_savefpr_19:  stfd f19,-104(r1)
_savefpr_20:  stfd f20,-96(r1)
_savefpr_21:  stfd f21,-88(r1)
_savefpr_22:  stfd f22,-80(r1)
_savefpr_23:  stfd f23,-72(r1)
_savefpr_24:  stfd f24,-64(r1)
_savefpr_25:  stfd f25,-56(r1)
_savefpr_26:  stfd f26,-48(r1)
_savefpr_27:  stfd f27,-40(r1)
_savefpr_28:  stfd f28,-32(r1)
_savefpr_29:  stfd f29,-24(r1)
_savefpr_30:  stfd f30,-16(r1)
_savefpr_31:  stfd f31,-8(r1)
              std  r0, 16(r1)
              blr


_restfpr_14:  lfd  f14,-144(r1)
_restfpr_15:  lfd  f15,-136(r1)
_restfpr_16:  lfd  f16,-128(r1)
_restfpr_17:  lfd  f17,-120(r1)
_restfpr_18:  lfd  f18,-112(r1)
_restfpr_19:  lfd  f19,-104(r1)
_restfpr_20:  lfd  f20,-96(r1)
_restfpr_21:  lfd  f21,-88(r1)
_restfpr_22:  lfd  f22,-80(r1)
_restfpr_23:  lfd  f23,-72(r1)
_restfpr_24:  lfd  f24,-64(r1)
_restfpr_25:  lfd  f25,-56(r1)
_restfpr_26:  lfd  f26,-48(r1)
_restfpr_27:  lfd  f27,-40(r1)
_restfpr_28:  lfd  f28,-32(r1)
_restfpr_29:  lfd  f29,-24(r1)
_restfpr_29:  ld   r0, 16(r1)
              lfd  f29,-24(r1)
              mtlr r0
              lfd  f30,-16(r1)
              lfd  f31,-8(r1)
              blr
_restfpr_30:  lfd  f30,-16(r1)
_restfpr_31:  ld   r0, 16(r1)
              lfd  f31,-8(r1)
              mtlr r0
              blr
```

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 64-bit PowerPC Architecture, only load and store instructions access memory. Because 64-bit PowerPC instructions cannot hold 64-bit addresses directly, a program normally computes an address into a register and accesses memory through the register.

It is possible to build addresses using absolute code which puts symbol addresses into instructions. However, the difficulty of building a 64-bit address means that 64-bit PowerPC code normally loads an address out of a memory location in the TOC section. Combining the TOC offset of the symbol with the TOC address in register r2 gives the absolute address of the TOC entry holding the desired address.

The following figures show sample assembly language equivalents to C language code. The @got syntax is explained above, in the section TOC Assembly Language Syntax.

Load and Store; variables are not in TOC:

```
C                               Assembly

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

dst = src;
                                ld  r6,src@got(r2)
                                ld  r7,dst@got(r2)
                                lwz r0,0(r6)
                                stw r0,0(r7)

ptr = &dst;
                                ld  r0,dst@got(r2)
                                ld  r7,ptr@got(r2)
                                std r0,0(r7)

*ptr = src;
                                ld  r6,src@got(r2)
                                ld  r7,ptr@got(r2)
                                lwz r0,0(r6)
                                ld  r7,0(r7)
                                stw r0,0(r7)
```

The next example shows the same code assuming that the variables are all stored in the TOC. Shared objects normally can not assume that globally visible variables are stored in the TOC. If they did, it would be impossible for the variable references to be redirected to overriding variables in the main program. Therefore, shared objects should normally always use the type of code shown above.

Load and Store; variables in TOC:

```
C                               Assembly

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

dst = src;
                                lwz r0,src@toc(r2)
                                stw r0,dst@toc(r2)

ptr = &dst;
                                la  r0,dst@toc(r2)
                                std r0,ptr@toc(r2)

*ptr = src;
                                lwz r0,src@toc(r2)
                                ld  r7,ptr@toc(r2)
                                stw r0,0(r7)
```

Function Calls

Programs use the 64-bit PowerPC bl instruction to make direct function calls. The bl instruction must be followed by a nop instruction. For PowerOpen compatibility, the nop instruction must be:

```
ori  r0,r0,0
```

For PowerOpen compatibility, the link editor must also accept these instructions as valid nop instructions:

```
cror 15,15,15
cror 31,31,31
```

In a relocatable object file, a direct function call should be made to the function entry point, which is a symbol beginning with dot (.). See the Section called *Function Descriptors* for more information.

When the link editor is creating an executable or shared object, and it sees a function call followed by a nop instruction, it determines whether the caller and the callee share the same TOC. If they do, it leaves the nop instruction unchanged. If they do not, the link editor constructs a linkage function. The linkage function loads the TOC register with the callee TOC and branches to the callee entry point. The link editor modifies the bl instruction to branch to the linkage function, and modifies the nop instruction to be

```
ld   r2,40(r1)
```

This will reload the TOC register from the TOC save area after the callee returns.

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.

If the callee is in a different shared object, a similar procedure of linkage code and a modified nop instruction is used. In this case, the dynamic linker must complete the link by filling in the function descriptor at run time. See the Section called *Procedure Linkage Table* in Chapter 5 for more details.

Here is an example of the assembly code generated for a function call:

```
C                              Assembly

extern void func (void);
func ();
                               bl   .func
                               ori  r0,r0,0
```

```
Here is an example of how the link editor transforms this code if the
callee has a different TOC than the caller:
```

```
C                              Assembly

extern void func (void);
func ();
                               bl   <linkage_for_func>
                               ld   r2,40(r1)
```

Here is an example of the linkage code created by the link editor. Remember that func@got@plt contains the address of the procedure linkage entry for func, which is a function descriptor. The function descriptor holds the addresses of the function entry point and the function TOC base.

```
<linkage_for_func>:
    ld    r12,func@got@plt(r2)
```

```
    std   r2,40(r1)
    ld    r0,0(r12)
    ld    r2,8(r12)
    mtctr r0
    bctr
```

The value of a function pointer is the address of the function descriptor, not the address of the function entry point itself.

```
C                               Assembly
extern void func (void);
extern void (*ptr) (void);
ptr = func;
                                ld    r6,func@got(r2)
                                ld    r7,ptr@got(r2)
                                std   r6,0(r7)

(*ptr) ();
                                ld    r6,ptr@got(r2)
                                ld    r6,0(r6)
                                ld    r0,0(r6)
                                std   r2,40(r1)
                                mtctr r0
                                ld    r2,8(r6)
                                bctrl
                                ld    r2,40(r1)
```

Since most of the code sequence used for a call through a pointer is the same no matter what function pointer is being used, it is also possible to do it by calling a function with an unusual calling convention provided by a library. With this approach, efficiency requires that the function be linked in directly, and not come from a shared library. The PowerOpen ABI uses a function named ._ptrgl for this purpose, passing the function pointer value in r11, and it is recommended that this name and calling convention be used as well when using this approach under ELF.

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.

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

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 example uses several simplifying conventions to hide irrelevant details:

- The selection expression resides in r12, and is of type int.
- The case label constants begin at zero.
- The case labels, the default, and the address table use assembly names .Lcasei, .Ldef, and .Ltab, respectively.

```
C                               Assembly
switch (j)
  {
```

```
  case 0:
    ...
  case 1:
    ...
  case 3:
    ...
  default:
    ...
  }
```

```
                                cmplwi  r12,4
                                bge     .Ldef
                                bl      .L1
                            .L1:
                                slwi    r12,2
                                mflr    r11
                                addi    r12,r12,.Ltab-.L1
                                add     r0,r12,r11
                                mtctr   r0
                                bctr
                            .Ltab:
                                b       .Lcase0
                                b       .Lcase1
                                b       .Ldef
                                b       .Lcase3
```

Dynamic Stack Space Allocation

Unlike some other languages, C does not need dynamic stack allocation within a stack frame. 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.

Here is the stack frame before dynamic stack allocation:

```
High address

          +-> Back chain
          |   Floating point register save area
          |   General register save area
          |   Local variable space
          |   Parameter save area    (SP + 48)
          |   TOC save area          (SP + 40)  --+
          |   link editor doubleword (SP + 32)    |
          |   compiler doubleword    (SP + 24)    |--stack frame header
          |   LR save area           (SP + 16)    |
          |   CR save area           (SP + 8)     |
SP  --->  +-- Back chain             (SP + 0)   --+

Low address
***Is there an example missing here?***
```

Here is the stack frame after dynamic stack allocation:

```
High address

          +-> Back chain
          |   Floating point register save area
          |   General register save area
          |   Local variable space
          |   -- Old parameter save area, now allocated space
          |   -- Old stack frame header, now allocated space
```

```
          |   -- More newly allocated space
          |   New parameter save area    (SP + 48)
          |   New TOC save area          (SP + 40)
          |   New link editor doubleword (SP + 32)
          |   New compiler doubleword    (SP + 24)
          |   New LR save area           (SP + 16)
          |   New CR save area           (SP + 8)
SP  --->  +-- New Back chain             (SP + 0)

Low address
***Is there an example missing here?***
```

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 save area is reserved for arguments passed in calls to other functions. See the Section called *Parameter Passing* for more information. 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 save area begin at a fixed offset (48) from the stack pointer, so this area must move when dynamic stack allocation occurs.

The stack frame header must also be at a fixed offset (0) from the stack pointer, so this area must also move when dynamic stack allocation occurs.

Data in the parameter save 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 save 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 quadword 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 stdu rS,-length(r1) if the length is less than 32768 bytes, or by using stdux rS,r1,rspace, where rS is the contents of the back chain word and rspace contains the (negative) rounded number of bytes to be allocated.

Note: It is only strictly necessary to copy the back chain. The information in the parameter save area is recreated for each function call. The information in the stack frame header, other than the back chain, is only used by a called function. In some cases, a compiler may need to copy the TOC save area as well, depending upon precisely how it generates linkage code.

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.

DWARF Definition

DWARF Release Number

This section defines the Debug With Arbitrary Record Format (DWARF) debugging format for the 64-bit PowerPC processor family. The 64-bit 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 Chapter 1.

The DWARF definition requires some machine-specific definitions. The register number mapping needs to be specified for the 64-bit PowerPC registers. In addition, the DWARF Version 2 specification requires processor-specific address class codes to be defined.

DWARF Register Number Mapping

This table outlines the register number mapping for the 64-bit PowerPC processor family. Note that for all special purpose registers, the number is simply 100 plus the SPR register number, as defined in the 64-bit PowerPC Architecture. Registers with an asterisk before their name are MPC601 chip-specific and are not part of the generic 64-bit PowerPC chip architecture.

Register Name	Number	Abbreviation
General Register 0-31	0-31	r0-r31
Floating Register 0-31	32-63	f0-f31
Condition Register	64	CR
Floating-Point Status and Control Register	65	FPSCR
* MQ Register	100	MQ or SPR0
Fixed-Point Exception Register	101	XER or SPR1
* Real Time Clock Upper Register	104	RTCU or SPR4

* Real Time Clock Lower Register	105	RTCL or SPR5
Link Register	108	LR or SPR8
Count Register	109	CTR or SPR9

For kernel debuggers, the mapping for all privileged registers is also defined in this table.

Register Name	Number	Abbreviation
Machine State Register	66	MSR
Segment Register 0-15	70-85	SR0-SR15
Data Storage Interrupt Status Register	118	DSISR or SPR18
Data Address Register	119	DAR or SPR19
Decrementer	122	DEC or SPR22
Storage Description Register 1	125	SDR1 or SPR25
Machine Status Save/Restore Register 0	126	SRR0 or SPR26
Machine Status Save/Restore Register 1	127	SRR1 or SPR27
Software-use Special Purpose Register 0	372	SPRG0 or SPR272
Software-use Special Purpose Register 1	373	SPRG1 or SPR273
Software-use Special Purpose Register 2	374	SPRG2 or SPR274
Software-use Special Purpose Register 3	375	SPRG3 or SPR275
Address Space Register	380	ASR or SPR280
External Access Register	382	EAR or SPR282
Time Base	384	TB or SPR284
Time Base Upper	385	TBU or SPR285
Processor Version Register	387	PVR or SPR287
Instruction BAT Register 0 Upper	628	IBAT0U or SPR528
Instruction BAT Register 0 Lower	629	IBAT0L or SPR529
Instruction BAT Register 1 Upper	630	IBAT1U or SPR530
Instruction BAT Register 1 Lower	631	IBAT1L or SPR531

Instruction BAT Register 2 Upper	632	IBAT2U or SPR532
Instruction BAT Register 2 Lower	633	IBAT2L or SPR533
Instruction BAT Register 3 Upper	634	IBAT3U or SPR534
Instruction BAT Register 3 Lower	635	IBAT3L or SPR535
Data BAT Register 0 Upper	636	DBAT0U or SPR536
Data BAT Register 0 Lower	637	DBAT0L or SPR537
Data BAT Register 1 Upper	638	DBAT1U or SPR538
Data BAT Register 1 Lower	639	DBAT1L or SPR539
Data BAT Register 2 Upper	640	DBAT2U or SPR540
Data BAT Register 2 Lower	641	DBAT2L or SPR541
Data BAT Register 3 Upper	642	DBAT3U or SPR542
Data BAT Register 3 Lower	643	DBAT3L or SPR543
* Hardware Implementation Register 0	1108	HID0 or SPR1008
* Hardware Implementation Register 1	1109	HID1 or SPR1009
* Hardware Implementation Register 2	1110	HID2 or IABR or SPR1010
* Hardware Implementation Register 5	1113	HID5 or DABR or SPR1013
* Hardware Implementation Register 15	1123	HID15 or PIR or SPR1023

The 64-bit PowerPC processor family defines the address class codes described in the following table:

Code	Value	Meaning
ADDR_none	0	No class specified

Chapter 4. Object Files

ELF Header

For file identification in e_ident, the 64-bit PowerPC processor family requires the values shown below:

```
e_ident[EI_CLASS]   ELFCLASS64   For all 64-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. Since the 64-bit PowerPC processor family defines no flags, this member contains zero.

Processor identification resides in the ELF header's e_machine member, and must have the value 21, defined as the name EM_PPC64.

The e_entry field in the ELF header holds the address of a function descriptor. See Function Descriptors in chapter 3. This function descriptor supplies both the address of the function entry point and the initial value of the TOC pointer register.

Special Sections

Various sections hold program and control information. The sections listed in the following table are used by the system and have the types and attributes shown.

```
Name          Type               Attributes

.glink        SHT_PROGBITS       SHF_ALLOC + SHF_EXECINSTR
.got          SHT_PROGBITS       SHF_ALLOC + SHF_WRITE
.toc          SHT_PROGBITS       SHF_ALLOC + SHF_WRITE
.tocbss       SHT_NOBITS         SHF_ALLOC + SHF_WRITE
.plt          SHT_NOBITS         SHF_ALLOC + SHF_WRITE
```

Note: The .plt section on the 64-bit PowerPC is of type SHT_NOBITS, not SHT_PROGBITS as on most other processors.

Special sections are described below.

```
Name     Description

.glink   This section may be used to hold the global linkage table which
         aids the procedure linkage table. See Procedure Linkage Table
         in Chapter 5 for more information.

.got     This section may be used to hold the Global Offset Table, or
         GOT.  See The Toc Section and Coding Examples in Chapter 3
         and Global Offset Table in Chapter 5 for more information.

.toc     This section may be used to hold the initialized Table of
         Contents, or TOC.  See TOC, below, The Toc Section and Coding
         examples in Chapter 3 and Global Offset Table in Chapter 5
         for more information.

.tocbss  This section may be used to hold the uninitialized portions
         of the TOC.  This data may also be stored as zero-initialized
         data in a .toc section.

.plt     This section holds the procedure linkage table. See Procedure
         Linkage Table in Chapter 5 for more information.
```

Note: Ttools which support this ABI are not required to use these sections precisely as defined here, and indeed are not required to use them at all. The true use of a section is defined by the relocation information and by the code which refers to it. However, if tools use these sections, they are required to give them the types and attributes specified in the above table.

TOC

The Table of Contents, or TOC, is part of the data segment of an executable program.

This sections describes a typical layout of the TOC in an executable file or shared object. Particular tools need not follow this layout as specified here.

The TOC typically contains data items within the .got, .toc and .tocbss sections, which can be addressed with 16-bit signed offsets from the TOC base. The TOC base is typically the first address in the TOC plus 0x8000, thus permitting a full 64 Kbyte TOC. The .got section is typically created by the link editor based on @got relocations. The .toc and .tocbss sections are typically included from relocatable object files referenced during the link.

The TOC may straddle the boundary between initialized and uninitialized data in the data segment. The usual order of sections in the data segment, some of which may be empty, is:

```
.data
.got
.toc
.tocbss
.plt
```

The link editor may create multiple TOC sections, as specified in the Section called *The TOC section* in Chapter 3. In such a case, the .got and .toc sections will be repeated as necessary, possibly renamed to preserve unique section names. Any occurrence of .tocbss in a TOC section other than the last one will be converted into a .toc section initialized to contain zero bytes.

Compilers may generate "short-form," one-instruction references for all data items that are in the TOC section for the object file being compiled. Such references are relative to the TOC pointer register, r2, which always holds the base of the TOC section for the object file.

In a shared object, only data items with local (non-global) scope may be addressed via the TOC pointer. Global data items must be addressed via the GOT, even if they appear in a .toc or .tocbss section.

A compiler which places some data items in the TOC must provide an option to avoid doing so in a particular compilation.

Symbol Table

Symbol Values

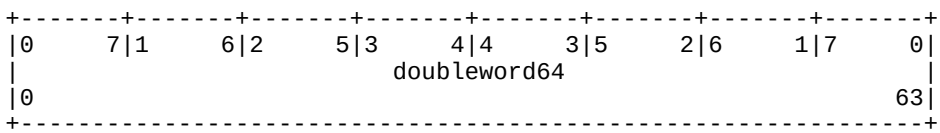
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for the file will contain an entry for that symbol. The st_shndx member of that symbol table entry contains SHN_UNDEF. This informs the dynamic linker that the symbol definition 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 function descriptor provided

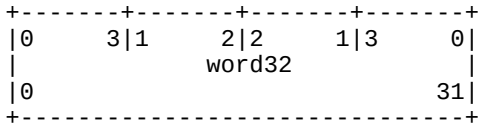
by that procedure linkage table entry. Otherwise, the st_value member contains zero. This procedure linkage table entry address is used by the dynamic linker in resolving references to the address of the function. See the Section called *Function Addresses* in Chapter 5 for details.

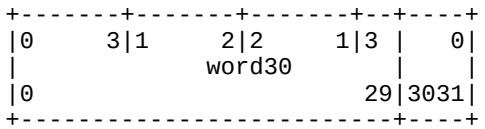
Relocation

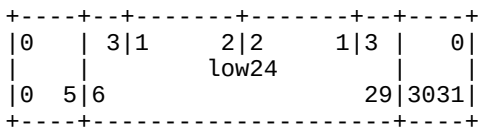
Relocation Types

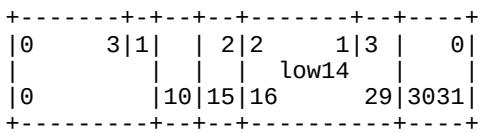
Relocation entries describe how to alter the instruction and data relocation fields shown below. Bit numbers appear in the lower box corners; little-endian byte numbers appear in the upper right box corners; big-endian numbers appear in the upper left box corners.

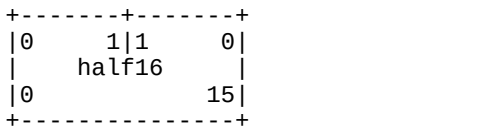


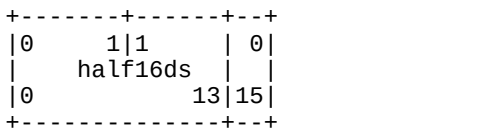












doubleword64 This specifies a 64-bit field occupying 8 bytes, the alignment of which is 8 bytes unless otherwise specified.

word32 This specifies a 32-bit field occupying 4 bytes, the alignment of which is 4 bytes unless otherwise specified.

word30 This 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 This 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 This 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 This specifies a 16-bit field occupying 2 bytes with 2-byte alignment (for example, the immediate field of an "Add Immediate" instruction).

half16ds Similar to half16, but really just 14 bits since the two least significant bits must be zero, and are not really part of the field. (Used by for example the ldu instruction.)

Calculations in the relocation table 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.

Some relocations use high adjusted values. These are the most significant bits, adjusted so that adding the low 16 bits will perform the correct calculation of the address accounting for signed arithmetic. This is to support using the low 16 bits as a signed offset when loading the value. For example, a value could be loaded from an absolute 64 bit address SYM as follows:

```
lis   r3,SYM@highesta
ori   r3,SYM@highera
sldi  r3,r3,32
oris  r3,r3,SYM@ha
ld    r4,SYM@l(r3)
```

The adjusted forms mean that this will work correctly even if SYM@l is negative when interpreted as a signed 16 bit number. Compare this to building the same 64 bit address using ori, in which case the adjusted forms are not used:

```
lis   r3,SYM@highest
ori   r3,SYM@higher
sldi  r3,r3,32
oris  r3,r3,SYM@h
ori   r3,r3,SYM@l
ld    r4,0(r3)
```

These code samples are not meant to encourage people to write code which builds absolute 64 bit addresses in this manner. It is normally better to use position independent code. However, this ABI does make this usage possible when it is required.

Relocations applied to executable or shared object files are similar and accomplish the same result. The following notations are used in the relocation table:

A Represents the addend used to compute the value of the

```
     relocatable field.

B    Represents the base address at which a shared object 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 Coding Examples in Chapter 3 and Global Offset
     Table in Chapter 5 for more information.

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
     Procedure Linkage Table in Chapter 5 for more information.

M    Similar to G, except that the address which is stored may be the
     address of the procedure linkage table entry for the symbol.

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.
```

Relocation entries apply to halfwords, words, or doublewords. In all cases, 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. The 64-bit PowerPC family uses only the Elf32_Rela relocation entries with explicit addends. For the 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 the relocation table:

- "+" and "-" denote 64-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 in which the names contain "32", the upper 32 bits of the value computed must be the same. For relocation types in which the names contain "14" or "16," the upper 49 bits of the value computed before shifting must all be the same. For relocation types whose names contain "24," the upper 39 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.
- #lo(value) denotes the least significant 16 bits of the indicated value:

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

- #hi(value) denotes bits 16 through 31 of the indicated value:

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

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

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

- #higher(value) denotes bits 32 through 47 of the indicated value.

```
#higher(x) = ((x >> 32) & 0xffff)
```

- #highera(value) denotes bits 32 through 47 of the indicated value, compensating for #lo() being treated as a signed number:

```
#highera(x) =
  (((x >> 32) + (((x & 0xffff8000) == 0xffff8000) ? 1 : 0)) & 0xffff)
```

- #highest(value) denotes bits 48 through 63 of the indicated value:

```
#highest(x) = ((x >> 48) & 0xffff)
```

- #highesta(value) denotes bits 48 through 63 of the indicated value, compensating for #lo being treated as a signed number:

```
#highesta(value) =
  (((x >> 48) + (((x & 0xffffffff8000) == 0xffffffff8000) ? 1 : 0)) & 0xffff)
```

- Reference in a calculation to the value G implicitly creates a GOT entry for the indicated symbol.
- .TOC. refers to the TOC base of the TOC section for the object being relocated. See the Section called *TOC* for additional information. The dynamic linker does not have this information, and hence relocation types that refer to .TOC. may only appear in relocatable object files, not in executables or shared objects.

Name	Value	Field	Calculation
R_PPC64_NONE	0	none	none
R_PPC64_ADDR32	1	word32*	S + A
R_PPC64_ADDR24	2	low24*	(S + A) >> 2
R_PPC64_ADDR16	3	half16*	S + A
R_PPC64_ADDR16_LO	4	half16	#lo(S + A)
R_PPC64_ADDR16_HI	5	half16	#hi(S + A)
R_PPC64_ADDR16_HA	6	half16	#ha(S + A)
R_PPC64_ADDR14	7	low14*	(S + A) >> 2
R_PPC64_ADDR14_BRTAKEN	8	low14*	(S + A) >> 2
R_PPC64_ADDR14_BRNTAKEN	9	low14*	(S + A) >> 2
R_PPC64_REL24	10	low24*	(S + A - P) >> 2
R_PPC64_REL14	11	low14*	(S + A - P) >> 2
R_PPC64_REL14_BRTAKEN	12	low14*	(S + A - P) >> 2
R_PPC64_REL14_BRNTAKEN	13	low14*	(S + A - P) >> 2
R_PPC64_GOT16	14	half16*	G + A
R_PPC64_GOT16_LO	15	half16	#lo(G + A)
R_PPC64_GOT16_HI	16	half16	#hi(G + A)
R_PPC64_GOT16_HA	17	half16	#ha(G + A)

R_PPC64_COPY	19	none	none
R_PPC64_GLOB_DAT	20	doubleword64	S + A
R_PPC64_JMP_SLOT	21	none	see below
R_PPC64_RELATIVE	22	doubleword64	B + A
R_PPC64_UADDR32	24	word32*	S + A
R_PPC64_UADDR16	25	half16*	S + A
R_PPC64_REL32	26	word32*	S + A - P
R_PPC64_PLT32	27	word32*	L + A
R_PPC64_PLTREL32	28	word32*	L + A - P
R_PPC64_PLT16_LO	29	half16	#lo(L + A)
R_PPC64_PLT16_HI	30	half16	#hi(L + A)
R_PPC64_PLT16_HA	31	half16	#ha(L + A)
R_PPC64_SECTOFF	33	half16*	R + A
R_PPC64_SECTOFF_LO	34	half16	#lo(R + A)
R_PPC64_SECTOFF_HI	35	half16	#hi(R + A)
R_PPC64_SECTOFF_HA	36	half16	#ha(R + A)
R_PPC64_ADDR30	37	word30	(S + A - P) >> 2
R_PPC64_ADDR64	38	doubleword64	S + A
R_PPC64_ADDR16_HIGHER	39	half16	#higher(S + A)
R_PPC64_ADDR16_HIGHERA	40	half16	#highera(S + A)
R_PPC64_ADDR16_HIGHEST	41	half16	#highest(S + A)
R_PPC64_ADDR16_HIGHESTA	42	half16	#highesta(S + A)
R_PPC64_UADDR64	43	doubleword64	S + A
R_PPC64_REL64	44	doubleword64	S + A - P
R_PPC64_PLT64	45	doubleword64	L + A
R_PPC64_PLTREL64	46	doubleword64	L + A - P
R_PPC64_TOC16	47	half16*	S + A - .TOC.
R_PPC64_TOC16_LO	48	half16	#lo(S + A - .TOC.)
R_PPC64_TOC16_HI	49	half16	#hi(S + A - .TOC.)
R_PPC64_TOC16_HA	50	half16	#ha(S + A - .TOC.)
R_PPC64_TOC	51	doubleword64	.TOC.
R_PPC64_PLTGOT16	52	half16*	M + A
R_PPC64_PLTGOT16_LO	53	half16	#lo(M + A)
R_PPC64_PLTGOT16_HI	54	half16	#hi(M + A)
R_PPC64_PLTGOT16_HA	55	half16	#ha(M + A)
R_PPC64_ADDR16_DS	56	half16ds*	(S + A) >> 2
R_PPC64_ADDR16_LO_DS	57	half16ds	#lo(S + A) >> 2
R_PPC64_GOT16_DS	58	half16ds*	(G + A) >> 2
R_PPC64_GOT16_LO_DS	59	half16ds	#lo(G + A) >> 2
R_PPC64_PLT16_LO_DS	60	half16ds	#lo(L + A) >> 2
R_PPC64_SECTOFF_DS	61	half16ds*	(R + A) >> 2
R_PPC64_SECTOFF_LO_DS	62	half16ds	#lo(R + A) >> 2
R_PPC64_TOC16_DS	63	half16ds*	(S + A - .TOC.) >> 2
R_PPC64_TOC16_LO_DS	64	half16ds	#lo(S + A - .TOC.) >> 2
R_PPC64_PLTGOT16_DS	65	half16ds*	(M + A) >> 2
R_PPC64_PLTGOT16_LO_DS	66	half16ds	#lo(M + A) >> 2

Figure 4-1. Relocation Table

> **Note:** Relocation values 18, 23 and 32 are not used. This is to maintain a correspondence to the relocation values used by the *32-bit PowerPC ELF ABI*.

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.

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.

Relocations 56-66 are to be used for instructions with a DS offset field (ld, ldu, lwa, std, stdu). ABI conformant tools should give an error for attempts to relocate an address to a value that is not divisible by 4.

Relocation types with special semantics are described below.

R_PPC64_GOT16*

These relocation types resemble the corresponding R_PPC64_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_PPC64_PLTGOT16*

These relocation types resemble the corresponding R_PPC64_GOT16* types, except that the address stored in the global offset table entry may be the address of an entry in the procedure linkage table. If the link editor can determine the actual value of the symbol, it may store that in the corresponding GOT entry. Otherwise, it may create an entry in the procedure linkage table, and store that address in the GOT entry; this permits lazy resolution of function symbols at run time. Otherwise, the link editor may generate a R_PPC64_GLOB_DAT relocation as usual.

R_PPC64_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_PPC64_GLOB_DAT

This relocation type resembles R_PPC64_ADDR64, 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_PPC64_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 the Section called *Procedure Linkage Table* in Chapter 5).

R_PPC64_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_PPC64_UADDR*

These relocation types are the same as the corresponding R_PPC64_ADDR* types, except that the datum to be relocated is allowed to be unaligned.

Chapter 5. Program Loading and Dynamic Linking

Program Loading

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 for the 64-bit PowerPC processor family segments are congruent modulo 64 Kbytes (0x10000) or larger powers of 2. Although 4096 bytes is currently the 64-bit PowerPC 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 0x10000.

It is normally desirable to put segments with different characteristics in separate 256 Mbyte portions of the address space, to give the operating system full paging flexibility in the 64-bit address space.

Here is an example of an executable file assuming an executable program linked with a base address of 0x10000000.

```
File Offset                             Virtual Address

0
            ELF header
            Program header table
            Other information
0x100                                   0x10000100
            Text segment
            . . .
            0x2be00 bytes
                                        0x1002beff
0x2bf00                                 0x2003bf00
            Data segment
            . . .
            0x4e00 bytes
                                        0x20040cff
0x30d00
            Other information
```

Here are possible corresponding program header segments:

```
Member      Text            Data

p_type      PT_LOAD         PT_LOAD
p_offset    0x100           0x2bf00
p_vaddr     0x10000100      0x2003bf00
p_paddr     unspecified     unspecified
p_filesz    0x2be00         0x4e00
p_memsz     0x2be00         0x5e24
p_flags     PF_R+PF_X       PF_R+PF_W
p_align     0x10000         0x10000
```

Note: The example addresses for the text and data segments are chosen for compatibility with AIX, and it is suggested, though not required, that tools supporting this ABI use similar addresses.

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 above, 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 above is shown here, assuming 4096 (0x1000) byte pages.

```
Text segment:
0x02000000
            Header padding
            0x100 bytes
0x02000100
            Text segment
            ...
            0x2be00 bytes
0x0202bf00
            Data padding
            0x100 bytes

Data segment:
0x0203b000
            Text padding
            0xf00 bytes
0x0203bf00
            Data segment
            ...
            0x4e00 bytes
0x02040d00
            Uninitialized data
            0x1024 bytes
0x02041d24
            Page padding
            0x2dc zero bytes
```

Figure 5-1. Virtual Address

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, most systems will maintain the "relative positions" of the segments. Any use of relative addressing between segments should be indicated by an appropriate dynamic relocation. If the dynamic linker does not maintain the relative position of segments at load time, it must be careful in its handling of R_PPC64_RELATIVE relocations, examining the relative address in order to determine the appropriate base address to use.

The following table shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

```
Source       Text        Data       Base Address
File         0x000200    0x02a400
Process 1    0x100200    0x12a400    0x100000
Process 2    0x200200    0x22a400    0x200000
Process 3    0x300200    0x32a400    0x300000
Process 4    0x400200    0x42a400    0x400000
```

Program Interpreter

The standard program interpreter is /usr/lib/ld.so.1.

Dynamic Linking

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.

DT_PLTGOT

This entry's d_ptr member gives the address of the first byte in the procedure linkage table.

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 the 64-bit PowerPC, 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 the Section called *Procedure Linkage Table* later in this chapter for more information.

Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. The global offset table, which is part of the TOC section, holds 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 TOC 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_PPC64_GLOB_DAT, referring to the global offset table within the TOC. 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 global offset table is part of the TOC section. Since different functions in a single executable or shared object may have different TOC sections, the global offset table may also be replicated, in whole or in part. Each instance of the global offset table will have its own set of relocations. The dynamic linker need not know about the replication; it simply processes all the relocations it is given.

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.

The global offset table normally resides in the ELF .got section in an executable or shared object.

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.

In this ABI, the address of a function is actually the address of a function descriptor. A reference to a function, other than a function call, will normally load the address of the function descriptor from the global offset table. The dynamic linker will ensure that for a given function, the same address is used for all references to the function from any global offset table. Thus, function address comparisons will work as expected.

When making a call to the function, the code may refer to the procedure linkage table entry, in order to permit lazy symbol resolution at run time. In order to support correct function address comparisons, the compiler should be careful to only generate references to the procedure linkage table entry for function calls. For any other use of a function, the compiler should use the real address.

When using the ELF assembler syntax, this means that the compiler should use the @got syntax, rather than the @got@plt syntax, if the function address is going to be used without being called.

Procedure Linkage Table

The procedure linkage table may be used to redirect function calls between the executable and a shared object or between different shared objects. Because all function calls on the 64-bit PowerPC are done via function descriptors, the procedure linkage table is simply a special case of a function descriptor which is filled in by the dynamic linker rather than the link editor.

The procedure linkage table is purely an optimization designed to permit lazy symbol resolution at run time. The link editor may generate R_PPC64_GLOB_DAT relocations for all function descriptors defined in other shared objects, and avoid generating a procedure linkage table at all.

The procedure linkage table is normally found in the .plt section in an executable or shared object. Its contents are 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 constraint:

- If the executable or shared object requires N procedure linkage table entries, the link editor shall reserve 3*(N+1) doublewords (24*(N+1) bytes). These doublewords will be used to hold function descriptors. When calling function i, the link editor arranges to use the function descriptor at byte 24 * i. The first procedure linkage table entry is 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 section 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_PPC64_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.

The dynamic linker will locate the symbol referenced by the R_PPC64_JMP_SLOT relocation. The value of the symbol will be the address of the function descriptor. The dynamic linker will copy these 24 bytes into the procedure linkage table entry.

The dynamic linker can resolve the procedure linkage table relocations lazily, resolving them only when they are needed. This can speed up program startup time.

The following code shows how the dynamic linker might initialize the procedure linkage table in order to provide lazy resolution:

```
.GLINK:
.GLINK0:
    ld      r2, 40(r1)
    addis   r12,r2,.PLT0@toc@ha
    addi    r12,r12,.PLT0@toc@l
    ld      r11,0(r12)
    ld      r2, 8(r12)
    mtctr   r11
    ld      r11,16(r12)
    bctr
.GLINK1:
    li      r0,0
    b       .GLINK0
.GLINKi:   # i <= 32768
    li      r0,i - 1
    b       .GLINK0
.GLINKN:   # N > 32768
    lis     r0,(N - 1) >> 16
    ori     r0,r0,(N - 1) & 0xffff
    b       .GLINK0

...

.PLT:
.PLT0:
    .quad   ld_so_fixup_func
    .quad   ld_so_toc
    .quad   ld_so_ident
.PLT1:
    .quad   .GLINK1
    .quad   0
    .quad   0
    ...
.PLTi:
```

```
    .quad   .GLINKi
    .quad   0
    .quad   0
    ...
.PLTN:
    .quad   .GLINKN
    .quad   0
    .quad   0
```

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

1. As shown above, each procedure linkage table entry I, as initialized by the link editor, transfers control to the corresponding glink entry I at .GLINKI. The instructions at .GLINKI loads a relocation index into r0 and branches to the common .GLINK0 code, the first entry in the GLINK table. For example, assume the program calls NAME, which uses the function descriptor at the label .PLTi. The function descriptor causes the program to branch to .GLINKi which loads i - 1 into r0 and branches to .GLINK0.
2. .GLINK0 loads three values from the PLT Reserve area allocated by the link editor and initialized by the dynamic linker. The first doubleword is the dynamic linker's lazy binding entry point. The second doubleword is the dynamic linker's own TOC anchor value. The third doubleword is an 8-byte identifier unique to the calling module which must be placed into r11 (normally the static chain), so that the dynamic linker can identify the object from which the call originated, and thereby located that object's relocation table. .GLINK0 then calls into the dynamic linker with the PLT index copied into r0 and the identifying information copied into r11.
3. The dynamic linker finds relocation entry i corresponding to the index in r0. It will have type R_PPC_JMP_SLOT, its offset will specify the address of .PLTi, and its symbol table index will reference NAME.
4. Knowing this, the dynamic linker finds the symbol's "real" value. It then copies the function descriptor into the code at .PLTi.
5. Subsequent executions of the procedure linkage table entry will transfer control directly to the function via the function descriptor at .PLTi, without invoking the dynamic linker.

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.

Lazy binding generally improves overall application performance because unused symbols do not incur the dynamic linking overhead. Nevertheless, 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 6. Libraries

This document does not specify any library interfaces.