M32R ELF Application Binary Interface Supplement 1.2

Kazuhiro Inaoka
Renesas Solutions Corp.

Kei Sakamoto
Renesas Technology Corp.

M32R ELF Application Binary Interface Supplement 1.2

by Kazuhiro Inaoka and Kei Sakamoto

1.2 Edition
Published Aug 26, 2004

Revision History

Revision 1.2 Aug 26, 2004 Revised by: Kazuhiro Inaoka, Renesas Solutions Corp.
Revise R_M32R_SDA16 and R_M32R_SDA16_RELA calculation.
Revision 1.1 Aug 19, 2004 Revised by: Kazuhiro Inaoka, Renesas Solutions Corp.
Revise relocation types information.
Revision 1.0 Jul 13, 2004 Revised by: Kei Sakamoto, Renesas Technology Corp.
Docbook formatted.

Table of Contents

List of Figures

Chapter 1. Introduction

1.1. The M32R Architecture and the System V ABI

The System V Application Binary Interface, or ABI, defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement the interfaces defined in the System V Interface Definition, Edition 4.

This document is a supplement to the generic System V ABI, and it contains information specific to System V implementations built on the M32R processor architecture. Together, these two specifications, the generic System V ABI and the M32R Architecture System V ABI Supplement (hereafter referred to as the M32R ABI), constitute a complete System V Application Binary Interface specification for systems that implement the processor architecture of the M32R microprocessors.

Note that this M32R ABI applies to any system built with the M32R processor chips.

1.2. How to Use the M32R ELF ABI Supplement

This document is a supplement to the generic System V ABI and contains information referenced in the generic specification that may differ when System V is implemented on different processors. Therefore, the generic ABI is the prime reference document, and this supplement is provided to fill gaps in that specification.

As with the System V ABI, this specification references other publicly- available reference documents, especially the Renesas M32R Software and Hardware Manual. All the information referenced by this supplement should be considered part of this specification, and just as binding as the requirements and data explicitly included here.

1.3. Evolution of the ABI Specification

The System V Application Binary Interface will evolve over time to address new technology and market requirements, and will be reissued at intervals of approximately three years. Each new edition of the specification is likely to contain extensions and additions that will increase the potential capabilities of applications that are written to confirm to the ABI.

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

These Level 1 and Level 2 classifications and qualifications apply to this Supplement, as well as to the generic specification. All components of the ABI and of this supplement have Level 1 support unless they are explicitly

labelled as Level 2.

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

- *M32R Software and Hardware Manual*, Renesas Technology Corp.
- *System V Interface Definition*, Issue 3.

Chapter 2. Software Installation

2.1. Physical Distribution Media and Formats

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

Chapter 3. Low Level System Information

3.1. Machine Interface

3.1.1. Processor Architecture

The Renesas M32R Software Manual defines the processor architecture. The architecture is hereafter referred to as the M32R architecture. Programs intended to execute directly on the processor use the instruction set, instruction encodings, and instruction semantics of the architecture. Three points deserve explicit mention.

- A program may assume all documented instructions exist.
- A program may assume all documented instructions work.
- A program may use only the instructions defined by the architecture.

In other words, from a program’s perspective, the execution environment provides a complete and working implementation of the M32R architecture.

This does not imply that the underlying implementation provides all instructions in hardware, only that the instructions perform the specified operations and produce the specified results. The ABI neither places performance constraints on systems nor specifies what instructions must be implementation in hardware. A software emulation of the architecture could conform to the ABI.

Some processors might support the M32R architecture as a subset, providing additional instructions or capabilities. Programs that use those capabilities explicitly do not conform to the M32R ABI. Executing those programs on machines without the additional capabilities gives undefined behavior.

3.1.2. Data Representation

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

3.1.3. Byte Ordering

The architecture defines an 8-bit byte, a 16-bit halfword, a 32-bit word, and a 64-bit doubleword. Byte ordering defines how the bytes that make up halfwords, words, and doublewords 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).

All M32R family processors support big-endian byte ordering and some processors of them support little-endian byte ordering too. 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 M32R 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

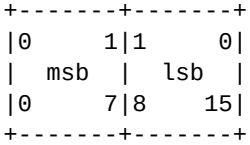


Figure 3-2. Bit and Byte Numbering in Words

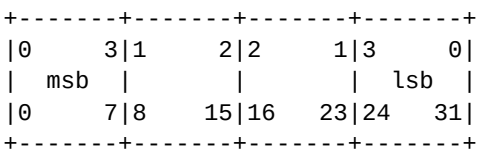
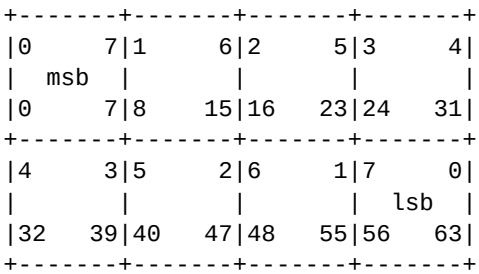


Figure 3-3. Bit and Byte Numbering in Doublewords



3.1.4. Fundamental Types

The following figure shows the correspondence between ANSI C's scalar types and the processor's.

Figure 3-4. Scalar Types

Type	C	sizeof	Alignment (bytes)	M32R Architecture
	char	1	1	signed byte
	signed char	1	1	
	unsigned char	1	1	unsigned byte

	short	2	2	signed halfword
	signed short			
	unsigned short	2	2	unsigned halfword
Integral	int			
	signed int			
	long	4	4	signed word
	signed long			
	enum			
	unsigned int	4	4	unsigned word
	unsigned long			
Pointer	any-type *	4	4	unsigned word
	any-type (*){}			
	float	4	4	single-precision(IEEE)
Floating-point	double	8	4	double-precision(IEEE)
	long double	8	4	double-precision(IEEE)

Note: The M32R architecture does not require doubleword alignment for double-precision values. Nevertheless, for data structure compatibility with other Renesas architectures, compilers may provide a method to align double-precision values on doubleword boundaries.

3.1.5. Aggregates and Unions

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

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

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

Figure 3-5. Structure Smaller Than a Word

```
struct {
  char c;
};
byte aligned, sizeof is 1
+-------+
|0     0|
```

```
|   c   |
+-------+
```

Figure 3-6. No Padding

```
struct {
  char  c;
  char  d;
  short s;
  int   n;
};
word aligned, sizeof is 8
big endian:

+-------+-------+-------+-------+
|0      |1      |2              |
|   c   |   d   |       s       |
+-------+-------+-------+-------+
|4                              |
|               n               |
+-------+-------+-------+-------+

little endian:

+-------+-------+-------+-------+
|              2|      1|      0|
|       s       |   d   |   c   |
+-------+-------+-------+-------+
|                              4|
|               n               |
+-------+-------+-------+-------+
```

Figure 3-7. Internal Padding

```
struct {
  char  c;
  short s;
};
halfword aligned, sizeof is 4
big endian:

+-------+-------+-------+-------+
|0      |1      |2              |
|   c   |  pad  |       s       |
+-------+-------+-------+-------+

little endian:

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

Figure 3-8. Internal and Tail Padding

```
struct {
 char      c;
 int       n;
 long long l;
 short     s;
};
word aligned, sizeof is 24
big endian:

+-------+-------+-------+-------+
|0      |1                      |
|   c   |          pad          |
+-------+-------+-------+-------+
|4                              |
|               n               |
+-------+-------+-------+-------+
|8                              |
|               l               |
+-------+-------+-------+-------+
|12                             |
|               l               |
+-------+-------+-------+-------+
|16             |18             |
|       s       |      pad      |
+-------+-------+-------+-------+
|20                             |
|             pad               |
+-------+-------+-------+-------+

little endian:

+-------+-------+-------+-------+
|                       1|      0|
|          pad          |   c   |
+-------+-------+-------+-------+
|                               4|
|               n               |
+-------+-------+-------+-------+
|                               8|
|               l               |
+-------+-------+-------+-------+
|                              12|
|               l               |
+-------+-------+-------+-------+
|             18|             16|
|      pad      |       s       |
+-------+-------+-------+-------+
|                              20|
|              pad              |
+-------+-------+-------+-------+
```

Figure 3-9. Union Allocation

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

+-------+-------+-------+-------+
|0      |1                      |
|   c   |          pad          |
+-------+-------+-------+-------+
|0              |2              |
|       s       |      pad      |
+-------+-------+-------+-------+
|0                              |
|               j               |
+-------+-------+-------+-------+

little endian:

+-------+-------+-------+-------+
|                      1|      0|
|          pad          |   c   |
+-------+-------+-------+-------+
|              2|              0|
|      pad      |       s       |
+-------+-------+-------+-------+
|                              0|
|               j               |
+-------+-------+-------+-------+
```

3.1.6. 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.

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
```

"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 32 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.

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.

Figure 3-10. Bit Numbering

```
0x01020304

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

Figure 3-11. Bit-field Allocation

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

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

little endian:

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

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

Figure 3-12. Boundary Alignment

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

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|
+-------+-----+--------+--------+
```

Figure 3-13. Doubleword Boundary Alignment

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

+-------------------------------+
|0                              |
|              i                |
|0                             31|
+-----------------------+-------+
|4                      |       |
|           i           |  pad  |
```

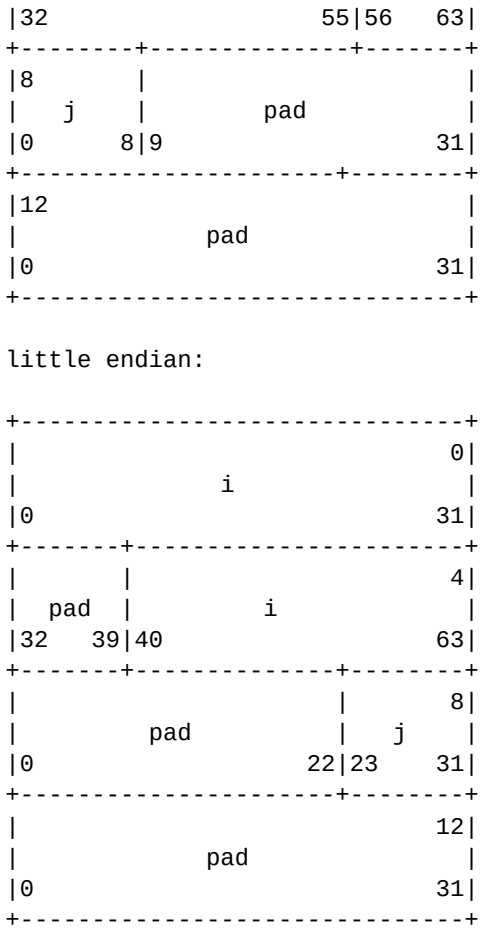


Figure 3-14. Storage Unit Sharing

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

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

little endian:

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

Figure 3-15. Union Allocation

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

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

little endian:

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

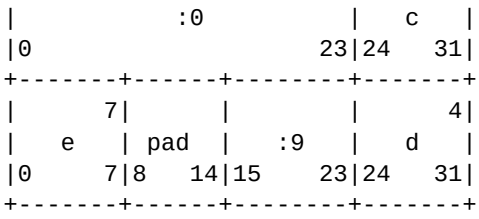
Figure 3-16. Unnamed bit-fields

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

+-------+-----------------------+
|0      |1                      |
|   c   |          :0           |
|0     7|8                     31|
+-------+--------+------+-------+
|4      |        |      |7      |
|   d   |   :9   | pad  |   e   |
|0     7|8     16|17  23|24   31|
+-------+--------+------+-------+

little endian:

+-----------------------+-------+
|                      1|      0|
```



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.

3.2. Function Calling Sequence

3.2.1. Registers and the Stack Frame

3.2.1.1. CPU Registers

```
Register Name       Use
(software name)
------------------------------------------------------------
r0 - r3             Used for passing arguments to functions.
                    Additional arguments are passed on the
                    stack (see below).  r0,r1 is also used
                    to return the result of function calls.
                    The values of these registers are not
                    preserved across function calls.

r4 - r7             Temporary registers for expression evaluation.
                    The values of these registers are not preserved
                    across function calls.

                    r6 is also reserved for use as a temp in the
                    PIC calling sequence (if ever necessary)
                    and may not be used in the function calling
                    sequence or prologue of functions.

                    r7 is also used as the static chain pointer
                    in nested functions (a GNU C extension)
                    and may not be used in the function calling
                    sequence or prologue of functions.

r8 - r11            Temporary registers for expression evaluation.
                    The values of these registers are preserved
                    across function calls.

r12                 Temporary register for expression evaluation.
                    Its value is preserved across function calls.
```

```
                    It is also reserved for use as potential
                    "global pointer".

r13 (fp)            Frame pointer.

r14 (lr)            Link register.
                    This register contains the return address
                    in function calls.

r15 (sp)            Stack pointer.
```

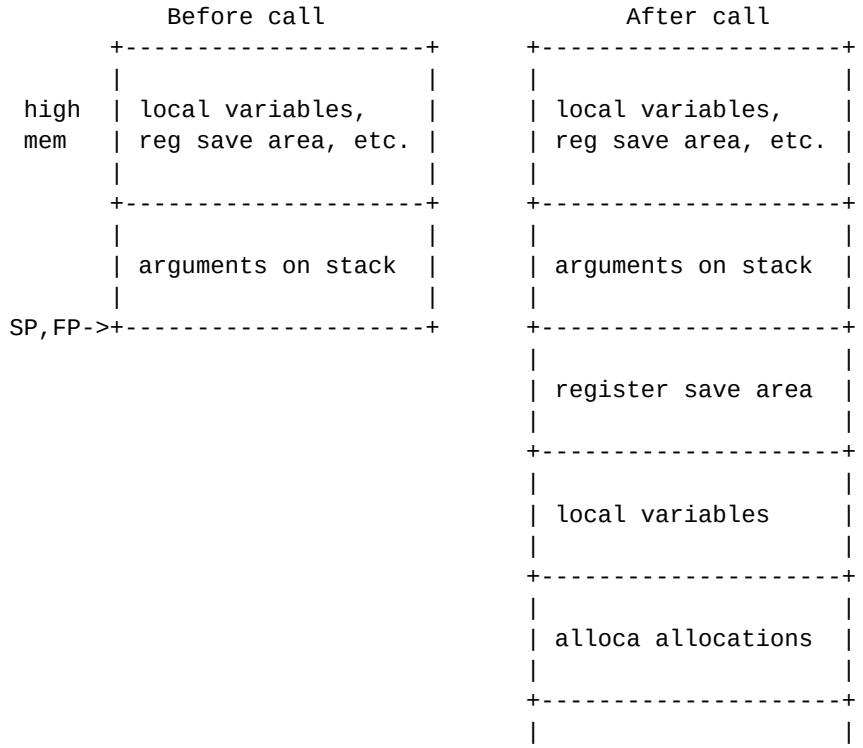
3.2.1.2. Special CPU Registers

```
condition bit       This is a 1 bit register that contains the
                    result of compare instructions.

accumulator         This is a 64 bit register that contains the
                    result of multiply/accumulate instructions.
```

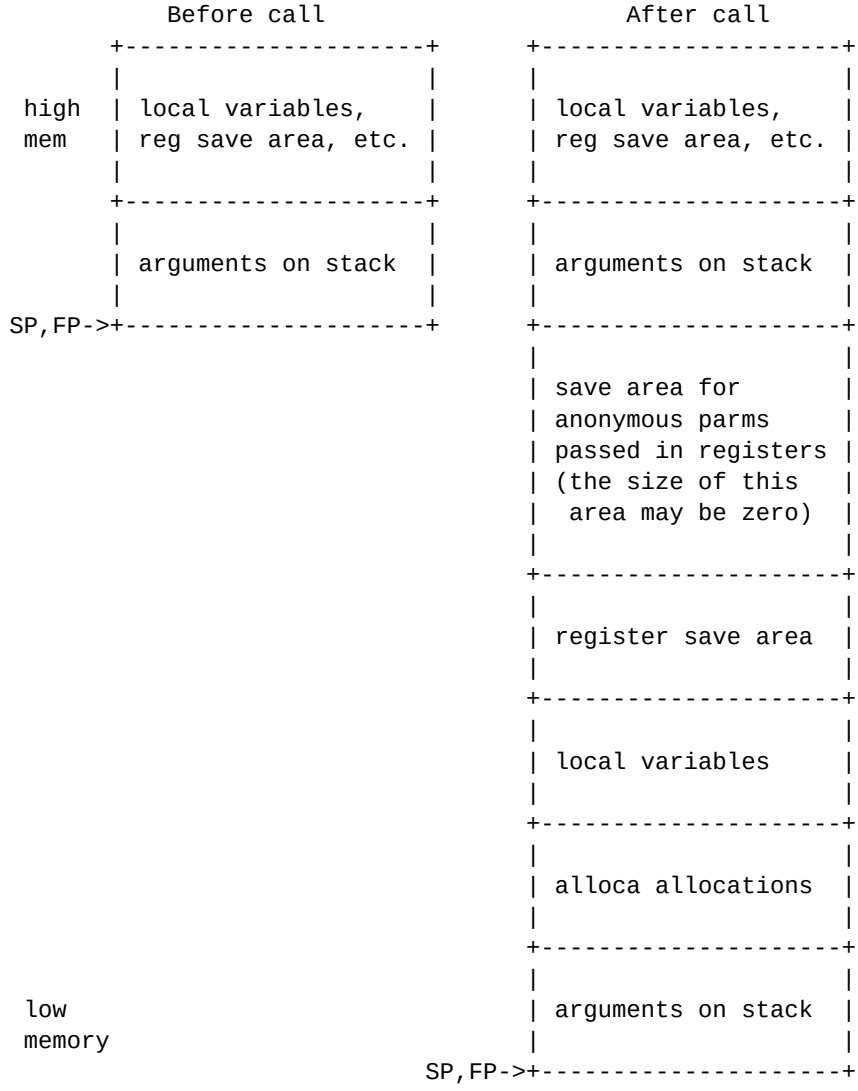
1. The stack grows downwards from high addresses to low addresses.
2. A leaf function need not allocate a stack frame if it doesn’t need one.
3. A frame pointer need not be allocated.
4. The stack pointer shall always be aligned to 4 byte boundaries.
5. The register save area shall be aligned to a 4 byte boundary.

Stack frames for functions that take a fixed number of arguments look like:



Note that FP points to the same location as SP.

Stack frames for functions that take a variable number of arguments look like:



3.2.1.3. Argument Passing

Arguments are passed to a function using first registers and then memory if the argument passing registers are used up. Each register is assigned an argument until all are used. Unused argument registers have undefined values on entry. The following rules must be adhered to.

1. Quantities of size 8 bytes or less are passed in registers if available, then memory. Larger quantities are passed by reference.

2. Arguments passed by reference are passed by making a copy on the stack and then passing a pointer to that copy.
3. If a data type would overflow the register arguments, then it is passed in registers and memory.

   e.g. A ‘long long’ data type passed in r3 would be passed in r3 and the first 4 bytes of the stack.

4. Arguments passed on the stack begin at ‘sp’ with respect to the caller.
5. Each argument passed on the stack is aligned on a 4 byte boundary.
6. Space for all arguments is rounded up to a multiple of 4 bytes.

3.2.1.4. Function Return Values

Integers, floating point values, and aggregates of 8 bytes or less are returned in register r0 (and r1 if necessary).

Aggregates larger than 8 bytes are returned by having the caller pass the address of a buffer to hold the value in r0 as an "invisible" first argument. All arguments are then shifted down by one. The address of this buffer is returned in r0.

3.2.1.5. Functions Returning Scalars or No Value

3.2.1.6. Functions Returning Structures or Unions

3.2.1.7. Integral and Pointer Arguments

3.2.1.8. Floating-Point Arguments

3.2.1.9. Structure and Union Arguments

3.3. Operating System Interface

3.3.1. Virtual Address Space

- Page Size
- Virtual Address Assignments
- Managing the Process Stack
- Coding Guidelines

3.3.2. Processor Execution Modes

3.3.3. Execution Interface

- Hardware Exception Types
- Software Trap Types

3.3.4. Process Initialization

- Special Registers
- Process Stack and Registers

3.4. Coding Examples

3.4.1. Code Model Overview

3.4.2. Function Calls

Absolute Direct Calls:

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

Absolute Indirect Calls:

```
C                               Assembly
extern void (*ptr) ();
```

```
(*ptr)();                       ld24 r4,#ptr
                                ld r5,@r4
                                jl r5
```

3.4.3. Switch Tables

Absolute code model:

```
C                               Assembly
void foo ()
{
...
switch (j)                      ld r4,[j]
{                               cmpui r4,#4
case 0:                         bnc .Ldef
        ...                     slli r4,#2
case 2:                         ld24 r5,#.Ltab
        ...                     add r4,r5
case 3:                         ld r4,@r4
        ...                     jmp r4
default:
        ...
}
...
}
                                .section .rodata
                        .Ltab:  .word .Lcase0
                                .word .Ldef
                                .word .Lcase2
                                .word .Lcase3
```

3.4.4. Position-Independent Function Prologue

3.4.5. Variable Argument List

3.4.6. Allocating Stack Space Dynamically

While C does not require dynamic stack allocation within a function, this ABI supports them. The stack pointer must be kept aligned to 4 byte boundaries.

Chapter 4. Object Files

4.1. ELF Header

For file identification e_ident, the M32R architecture requires the following values.

Position	Value
e_ident[BI_CLASS]	ELFCLASS32
e_ident[BI_DATA]	ELFDATA2MSB

Processor identification resides in the ELF header's e_machine member and must have the value EM_M32R.

The ELF header's e_flags member holds bit flags associated with the file. The M32R architecture defines no flags; so this member contains zero.

4.2. Special Sections

Various sections hold programs and control information. Sections in the list below are used by the system and have the indicated types and attributes.

Name	Type	Attributes
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.plt	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

```
-----

.got    This section holds the global offset table.  See Section 3.4
        and Section 5.2.2 for more information.

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

4.3. Symbol Table

4.3.1. Symbol Values

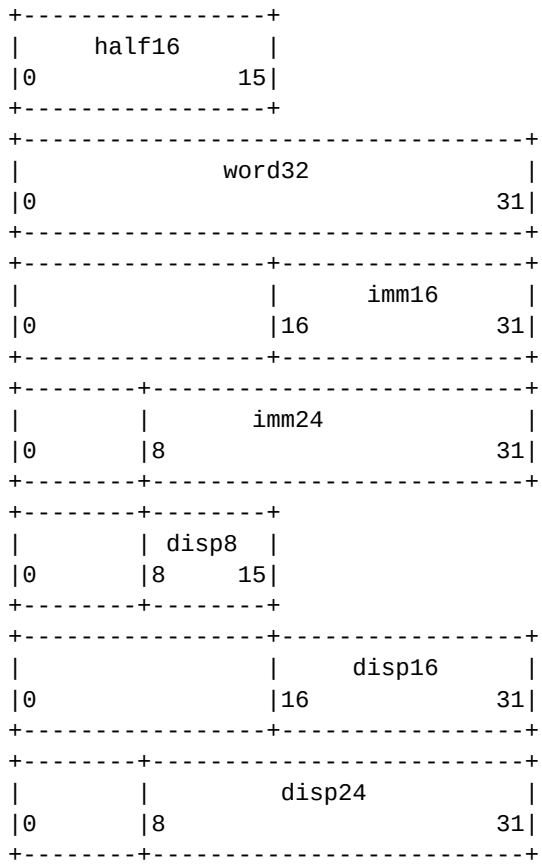
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for that file will contain an entry for that symbol. The st_shndx member of that symbol table entry contains SHN_UDEF. This signals to the dynamic linker that the symbol 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 non-zero, the value will contain the virtual address of the first instruction of that procedure linkage table entry. Otherwise, the st_value member contains zero. This procedure linkage table entry address is used by the dynamic linker in resolving references to the address of the function. See Section 5.2.3 for details.

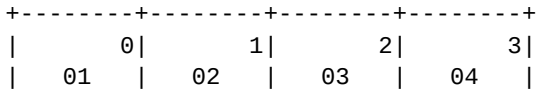
4.4. Relocation

4.4.1. 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.



word32 This specifies a 32-bit fields occupying 4 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the M32R architecture.



```
        0x01020304  |0       |        |        |      31|
                    +--------+--------+--------+--------+
```

Calculations below 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 decides how to combine and locate the input files, then updates the symbol values, and finally performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

```
A       This means the added used to compute the value of the relocatable
        field.

B       This means 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       This means the offset into the global offset table at which the
        address of the relocation entry's symbol will reside during execution.
        See Section 3.4 and Section 5.2.2 for more information.

GOT     This means the address of the global offset table.  See
        Section 3.4 and Section 5.2.2 for more information.

L       This means the place (section offset or address) of the procedure
        linkage table entry for a symbol.  A procedure linkage table entry
        redirects a function call to the proper destination.  The link
        editor builds the initial procedure linkage table, and the dynamic
        linker modifies the entries during execution.  See Section 5.2.4
        for more information.

P       This means the place (section offset or address) of the storage
        unit being relocated (computed using r_offset).

S       This means the value of the symbol whose index resides in the
        relocation entry.
```

A relocation entry's 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 M32R architecture uses only Elf32_Rel relocation entries, the field to be relocated holds the addend. In all cases, the addend and the computed result use the same byte order.

Figure 4-1. Relocation Table

```
    Name             Value      Field   Calculation
-------------------------------------------------------------------
R_M32R_NONE            0        none    none
R_M32R_16              1        half16  S+A                    ------
R_M32R_32              2        word32  S+A                       ^
R_M32R_24              3        imm24   (S+A)&0xFFFFFF            |
R_M32R_10_PCREL        4        disp8   ((S+A-P)>>2)&0xFF         |
R_M32R_18_PCREL        5        disp16  ((S+A-P)>>2)&0xFFFF       |
R_M32R_26_PCREL        6        disp24  ((S+A-P)>>2)&0xFFFFFF     |
R_M32R_HI16_ULO        7        imm16   ((S+A)>>16)            OLD TYPE
```

```
R_M32R_HI16_SLO      8        imm16   ((S+A)>>16)                 |
                                        or ((S+A+0x10000)>>16)  |
R_M32R_LO16          9        imm16   (S+A)&0xFFFF                |
R_M32R_SDA16         10       imm16   (S+A-_SDA_BASE_)&0xFFFF     |
R_M32R_GNU_VTINHERIT 11                                          V
R_M32R_GNU_VTENTRY   12                                       ------

R_M32R_16_RELA       33       half16  S+A
R_M32R_32_RELA       34       word32  S+A
R_M32R_24_RELA       35       imm24   (S+A)&0xFFFFFF
R_M32R_10_PCREL_RELA 36       disp8   ((S+A-P)>>2)&0xFF
R_M32R_18_PCREL_RELA 37       disp16  ((S+A-P)>>2)&0xFFFF
R_M32R_26_PCREL_RELA 38       disp24  ((S+A-P)>>2)&0xFFFFFF
R_M32R_HI16_ULO_RELA 39       imm16   ((S+A)>>16)
R_M32R_HI16_SLO_RELA 40       imm16   ((S+A)>>16)
                                        or ((S+A+0x10000)>>16)
R_M32R_LO16_RELA     41       imm16   (S+A)&0xFFFF
R_M32R_SDA16_RELA    42       imm16   (S+A-_SDA_BASE_)&0xFFFF
R_M32R_RELA_GNU_VTINHERIT 43
R_M32R_RELA_GNU_VTENTRY   44

R_M32R_GOT24         48       imm24   G+A-P
R_M32R_26_PLTREL     49       disp24  L+A-P
R_M32R_COPY          50       none    none
R_M32R_GLOB_DAT      51       word32  S
R_M32R_JMP_SLOT      52       word32  S
R_M32R_RELATIVE      53       word32  B+A
R_M32R_GOTOFF        54       word32  GOT-(S+A)
R_M32R_GOTPC24       55       word32  GOT+A-P
R_M32R_GOT16_HI_ULO  56       imm16   ((G+A-P)>>16)
R_M32R_GOT16_HI_SLO  57       imm16   ((G+A-P)>>16)
                                        or ((G+A-P+0x10000)>>16)
R_M32R_GOT16_LO      58       imm16   (G+A-P)&0xFFFF
R_M32R_GOTPC_HI_ULO  59       imm16   (GOT+A-P)>>16
R_M32R_GOTPC_HI_SLO  60       imm16   (GOT+A-P)>>16
                                        or ((GOT+A-P+0x10000)>>16)
R_M32R_GOTPC_LO      61       imm16   (GOT+A-P)&0xFFFF
R_M32R_GOTOFF_HI_ULO 62       imm16   (S+A-GOT)>>16
R_M32R_GOTOFF_HI_SLO 63       imm16   (S+A-GOT)>>16
                                        or (S+A-GOT+0x10000)>>16
R_M32R_GOTOFF_LO     64       imm16   (S+A-GOT)&0xFFFF
-----------------------------------------------------------------------
```

Some relocation types have semantics beyond simple calculation.

R_M32R_GOT24
R_M32R_GOT16_HI_ULO
R_M32R_GOT16_HI_SLO
R_M32R_GOT16_LO

This relocation type computes the distance from the base of the global offset table to the symbol's global offset table entry. It additionally instructs the link editor to build a global offset table.

```
ex1) ld24   r12,#label
ex2) seth   r12,#high(label)
     or3    r12,r12,#low(label)
ex3) seth   r12,#shigh(label)
```

```
add3   r12,r12,#low(label)
```

R_M32R_26_PLTREL

This relocation type computes the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a global offset table.

R_M32R_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_M32R_GLOB_DAT

This relocation type used to set a global offset table entry to the address of the specified symbol. The special relocation type allows one to determine the correspondence between symbols and global offset table entries.

R_M32R_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 to transfer control to the designated symbol's address (See Section 5.2.4 for more information).

R_M32R_RELATIVE

The link editor creates this relocation type for dynamic linking. Its offset member gives the location within a shred 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_M32R_GOTOFF
R_M32R_GOTOFF_HI_ULO
R_M32R_GOTOFF_HI_SLO
R_M32R_GOTOFF_LO

This relocation type computes the difference between a symbol's value and the address of the global offset table. It additionally instructs the link editor to build the global offset table.

R_M32R_GOTPC24
R_M32R_GOTPC_HI_ULO
R_M32R_GOTPC_HI_SLO
R_M32R_GOTPC_LO

These relocation types resemble R_M32R_26_PCREL, etc., except they use the address of the global offset table in their calculation. The symbol referenced in these relocations normally is _GLOBAL_OFFSET_TABLE_, which additionally instructs the link editor to build a global offset table.

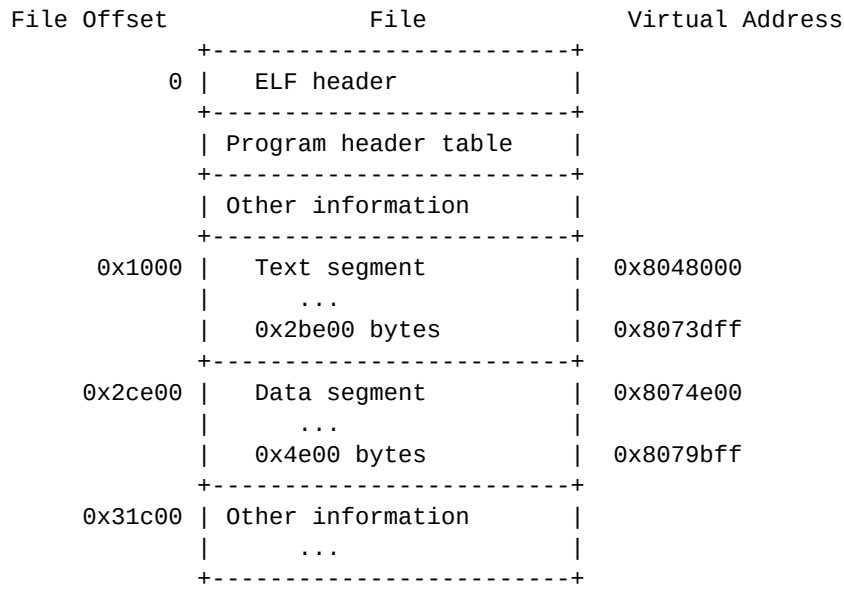
```
ex1) ld24   r12,#_GLOBAL_OFFSET_TABLE_
ex2) seth   r12,#high(_GLOBAL_OFFSET_TABLE_)
     or3    r12,r12,#low(_GLOBAL_OFFSET_TABLE_+4)
ex3) seth   r12,#shigh(_GLOBAL_OFFSET_TABLE_)
     add3   r12,r12,#low(_GLOBAL_OFFSET_TABLE_+4)
```

Chapter 5. Program Loading and Dynamic Linking

5.1. Program Loading

As the system creates or arguments 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 efficiently in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for the M32R architecture segments are congruent modulo 4KB (0x1000) or larger powers of 2. Because 4KB is the maximum page size, the files will be suitable for paging regardless of physical page size.



5.2. Dynamic Linking

5.2.1. Dynamic Section

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

DT_PLTGOT

On the M32R architecture, the entry's d_ptr member gives the address of the first entry in the global offset table. As mentioned below, the first three global offset table entries are reserved, and two are used to hold procedure linkage table information.

5.2.2. Global Offset Table

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

Initially, the global offset table holds information as required by its relocation entries [see "Relocation" in Chapter 4]. After the system creates memory segments for a loadable object file, the dynamic linker processes the relocation entries, some of which will be type R_M32R_GLOB_DAT referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory 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.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address 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 table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image. On the M32R architecture, entries one and two in the global offset table also are reserved. "Procedure Linkage Table" below describes them.

The system may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For the M32R architecture, the symbol _GLOBAL_OFFSET_TABLE_ may be used to access the table.

5.2.3. Function Addresses

References to the address of a function from an executable file and the shared objects associated with it might not resolve to the same value. References from within shared objects will normally be resolved by the dynamic linker to the virtual address of the function itself. References from within the executable file to a function defined

in a shared object will normally be resolved by the link editor to the address of the procedure linkage table entry for that function within the executable file.

To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry. [See "Symbol Values" in Chapter 4]. The dynamic linker treats such symbol table entries specially. If the dynamic linker is searching for a symbol, and encounters a symbol table entry for that symbol in the executable file, it normally follows the rules below.

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

Some relocations are associated with procedure linkage table entries. These entries are used for direct function calls rather than for references to function addresses. These relocations are not treated in the special way described above because the dynamic linker must not redirect procedure linkage table entries to point to themselves.

5.2.4. Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the M32R architecture, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position- independence and sharability of the program's text. Executable files and shared object files have separate procedure linkage tables.

Figure 5-1. Absolute Procedure Linkage Table (32-bit version)

```
.PLT0:  seth    r6,#high(.got+4)
        or3     r6,r6,#low(.got+4)
        ld      r4,@r6+          ; argument to the linker (id of GOT)
        ld      r6,@r6           ; dynamic linker address
        jmp     r6

.PLT1:  seth    r6,#shigh(.got+name1_GOT)
        ld      r6,@(low(.got+name1_GOT),r6)
        jmp     r6
        ld24    r5,#name1_dynamic_offset  ; arg to the linker (reloc offset)
        bra     .PLT0
        ...

.PLTn:  seth    r6,#shigh(.got+nameN_GOT)
```

```
        ld      r6,@(low(.got+nameN_GOT),r6)
        jmp     r6
        ld24    r5,#nameN_dynamic_offset  ; arg to the linker (reloc ofsset)
        bra     .PLT0
        ...
```

Note: PLT entry max-size is 20byte.

Figure 5-2. Position-Independent Procedure Linkage Table (32-bit version)

```
.PLT0:  ld      r4,@(4,r12)  ; argument to the linker (id of GOT) on stack
        ld      r6,@(8,r12)  ; dynamic linker address
        jmp     r6
        ...

.PLTn:  ld24    r6,#nameN_GOT
        add     r6,r12
        ld      r6,@r6
        jmp     r6
        ld24    r5,#nameN_dynamic_offset  ; arg to the linker (reloc offset)
        bra     .PLT0
        ...
```

Note: PLT entry max-size is 20byte.

Following the steps below, the dynamic linker and the program "cooperate" to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.
2. If the procedure linkage table is position-independent, the address of the global offset table must reside in R12. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file. Consequently, the calling function is responsible for setting the global offset table base register before calling the procedure linkage table entry.
3. For illustration, assume the program calls name1, which transfers control to the label .PLT1.
4. The first instruction jumps to the address in the global offset table entry for name1. Initially, the global offset table holds the address of the following ld24 instruction, not the real address of name1.
5. Consequently, the program loads a relocation offset (offset) into R4. The relocation offset is a 32-bit, non-negative byte offset into the relocation table. The designated relocation entry will have type R_M32R_JMP_SLOT, and its offset will specify the global offset table entry used in the previous jmp instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, name1 in this case.
6. After loading the relocation offset, the program then jumps to .PLT0, the first entry in the procedure linkage table. The ld instruction sets the value of the second global offset table entry (@(4,R12)) to R6, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry (@(8,R12)), which transfers control to the dynamic linker.

7. When the dynamic linker receives control, it refers to the arguments passed by the registers R4 and R5, looks at the designated relocation entry, finds the symbol’s value, stores the "real" address of name1 in its global offset table entry, and transfers control to the desired destination.
8. Subsequent executions of the procedure linkage table entry will transfer directly to name1, without calling the dynamic linker a second time. That is, the ld24 instruction at .PLT1 will transfer to name1, instead of "falling through" to the next ld24 instruction.

The LD_BIND_NOW environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, dynamic linker processes relocation entries of type R_M32R_JMP_SLOT during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

> **Note:** Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, 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.

5.2.5. Program Interpreter

There is one valid program interpreter for programs conforming to the M32R ABI: `/lib/ld-linux.so.2`.

Chapter 6. Libraries

This document does not specify any library interfaces.

Appendix A. GNU Free Documentation License

Version 1.1, March 2000

A.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.