DesignWare ARCv2 System V ABI Supplement User's Guide

Version 4092-007

Version 4092-007
November 2019

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
www.synopsys.com

CONTENTS

List of Tables

List of Figures

1 Introduction

The *System V Application Binary Interface* defines a linking interface for compiled application programs. The interface is described in two parts:

- The first part is the generic System V ABI, shared across all processor architectures.
- The second part is a processor specific supplement.

This document is the processor specific supplement for use with ELF on processor systems based on the ARCv2 instruction-set architecture.

This document is not a complete *System V Application Binary Interface Supplement*, because it does not define any OS library-interface information. Further, this ABI pertains primarily to C and assembly and contains only limited information on C++.

In the ARCv2 architecture, a processor can run in either of two modes: big-endian mode or little-endian mode. 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. An ARCv2 ABI-conforming system must support little-endian byte ordering. Accordingly, the ABI specification defines only the little-endian byte-ordering model. The ARCv2 ELF ABI is not the same as the preliminary ARC ABI published December 1999–April 2010.

Note: This ABI does not specify software installation, media, and formats.

1.1 How to Use the System V ABI Supplement

While the generic *System V ABI* is the prime reference document, this document contains ARCv2-specific implementation details, some of which supersede information in the generic ABI.

As with the System V ABI, this *Supplement* refers to other documents, especially the *ARCv2 Programmer's Reference Manual*, all of which should be considered part of this *ARCv2 ABI Supplement* and as binding as the requirements and data it explicitly includes.

1.1.1 Structure

This ABI Supplement consists of the following major divisions:

Chapter 2, *Low-Level System Information* describes the machine interface, byte ordering, data types, storage mapping, function calling sequence, registers, stack frame, function prolog and epilog, and function calls and branching.

Chapter 3, *Object Files* describes the ELF header, special sections, symbol table, small-data area, mapping variables to registers, and relocation types and fields.

Chapter 4, *Program Loading and Dynamic Linking* is of interest to UNIX-style operating systems, and describes how programs are loaded and dynamically linked, including the global offset table and procedure linkage table.

1.1.2 Terminology

Callee-saved *Callee-saved registers* (sometimes called *non-volatile registers*) hold values that are expected to be preserved across calls.

Caller-saved *Caller-saved registers* (sometimes called *volatile registers*) hold temporary values that are not expected to be preserved across calls.

Word Thirty-two bits of data, unless otherwise specified.

1.2 Evolution of the ABI Specification

Each new edition of *System V Application Binary Interface* is likely to contain extensions and additions that increases the potential capabilities of applications that are written to conform to the ABI.

1.3 Reference Documents

- *System V Interface Definition, Issue 3*
- *DWARF Debugging Information Format*, Version 4, 2010, Free Standards Group, DWARF Debugging Information Format Workgroup
- *ARCv2 Programmer's Reference Manual*

Table 1.1: Revision History

Version	Date	Description
4092-001	July 2015	Initial publication
4092-002	September 2015	Added information on overlay-related sections
4092-003	December 2015	• Added information on .vectors section • Specified that signed integral types are used by default for enums.
4092-004	June 2016	• Corrected `dispu7` to `disp7u` and documented the field. • Corrected `disp10` field name to `disp10u` and documented the field. • Clarified that With the exception of `word32`, all relocations with replacement fields in four-byte words must be written using Middle- Endian Storage. • Labeled `word32` fields *word32me* when they are subject to middle-endian storage. • Removed relocation type `R_ARC_SPE_SECTOFF` • Corrected calculations of relocation types: – `R_ARC_32_ME` – `R_ARC_N32_ME` – `R_AC_SECTOFF_S9` – `R_AC_SECTOFF_S9_1` – `R_AC_SECTOFF_S9_2` • Corrected field type of `R_ARC_AOM_TOKEN_ME` from `limm` to `word32me` • Further clarified explanation of `R_ARC_*_ME` relocation type • Noted that the ninth bit of the replacement field is not used for the following relocation types: – `R_AC_SECTOFF_U8` – `R_AC_SECTOFF_U8_1` – `R_AC_SECTOFF_U8_2` • Corrected various typographical errors.
4092-005	March 2018	• Made the relocation displacement figures bit-exact. • Clarified that the `LP_COUNT`, `r58`, and `r59` registers are accumulators and caller-saved registers. • Clarified that register `r25` is reserved by the EV6x processors • Clarified that the `r30` register is used as a scratch register • Clarified that the `r25` register is used for TLS by gcc • Clarified that when calling an external function, the compiler assumes that registers `r0` through `r12` and `r30` are trashed; and that `r13` through r29 are preserved. • Clarified that gcc reserves `r25` as Thread pointer if Thread local storage is enabled.

2 Low-Level System Information

2.1 Machine Interface

2.1.1 Processor Architecture

Programs intended to execute on ARCv2-based processors use the ARCv2 instruction set and the instruction encoding and semantics of the architecture.

Assume that all instructions defined by the architecture are neither privileged nor exist optionally and work as documented.

To conform to ARCv2 System V ABI, the processor must do the following:

- implement the instructions of the architecture,
- perform the specified operations,
- 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 can conform to the ABI.

Caution: Some processors might support optional or additional instructions or capabilities that do not conform to the ARCv2 ABI. Executing programs that use such instructions or capabilities on hardware that does not have the required additional capabilities results in undefined behavior.

2.1.2 Data Representation

Byte Ordering

The architecture defines an eight-bit byte, a 16-bit halfword, a 32-bit word, and a 64-bit double word. Byte ordering defines how the bytes that make up halfwords, words, and doublewords are ordered in memory.

Most-significant-byte (MSB) ordering, also called as “big-endian”, means that the most-significant byte is located in the lowest addressed byte position in a storage unit (byte 0).

Least-significant-byte (LSB) ordering, also called as “little-endian”, means that the least-significant byte is located in the lowest addressed byte position in a storage unit (byte 0).

ARCv2-based processors support either big-endian or little-endian byte ordering. However, this specification defines only the base-case little-endian (LSB) architecture.

Fig. 2.1 through Fig. 2.3 illustrate the conventions for bit and byte numbering within storage units of varying width. 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 byte numbers in the upper right corners, and bit numbers in the lower corners.

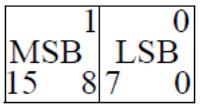

Fig. 2.1: Bit and Byte Numbering in Halfwords

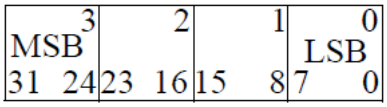

Fig. 2.2: Bit and Byte Numbering in Words

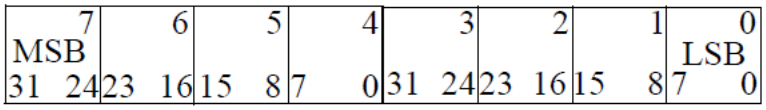

Fig. 2.3: Bit and Byte Numbering in Doublewords

Data Layout in Memory

ARCv2-based processors access data memory using byte addresses and generally require that all memory addresses be aligned as follows:

- 64-bit double-words are aligned to 32-bit word boundaries.
- 32-bit words are aligned to 32-bit word boundaries.
- 16-bit halfwords are aligned to 16-bit halfword boundaries.

Bytes have no specific alignment.

Sixty-Four-Bit Data

Fig. 2.4 shows the little-endian representation in byte-wide memory. If the ARCv2-based processor supports big-endian addressing, the data is stored in memory as shown in Fig. 2.5.

Thirty-Two-Bit Data

Fig. 2.6 shows the data representation in a general purpose register.

Fig. 2.7 shows the little-endian representation in byte-wide memory.

Fig. 2.8 shows the big-endian representation.

Sixteen-Bit Data

Fig. 2.9 shows the 16-bit data representation in a general purpose register.

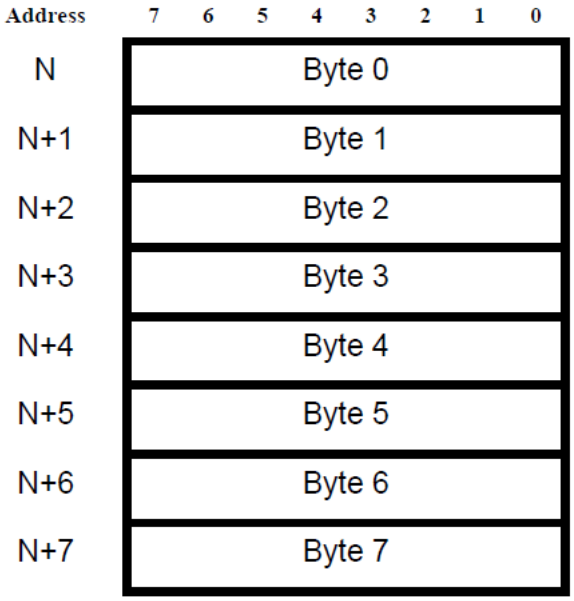

Fig. 2.4: Sixty-Four-Bit Register Data in Byte-Wide Memory, Little-Endian

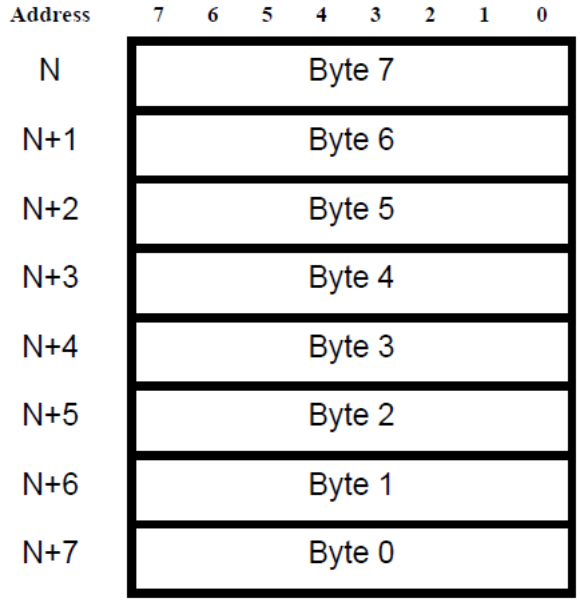

Fig. 2.5: Sixty-Four-Bit Register Data in Byte-Wide Memory, Big-Endian

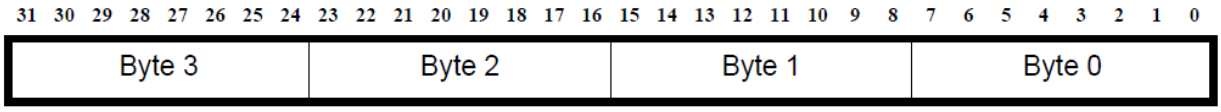

Fig. 2.6: Register Containing Thirty-Two-Bit Data

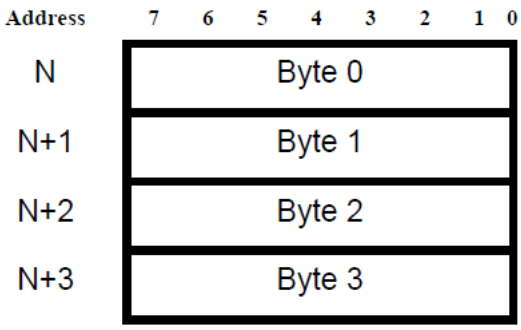

Fig. 2.7: Thirty-Two-Bit Register Data in Byte-Wide Memory, Little-Endian

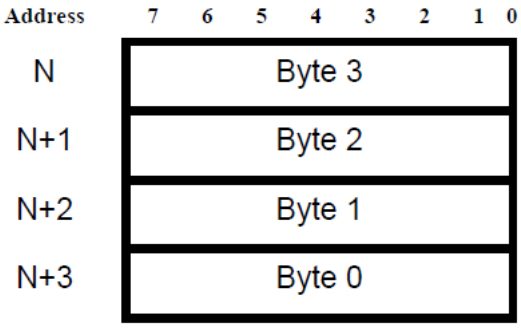

Fig. 2.8: Thirty-Two-Bit Register Data in Byte-Wide Memory, Big-Endian

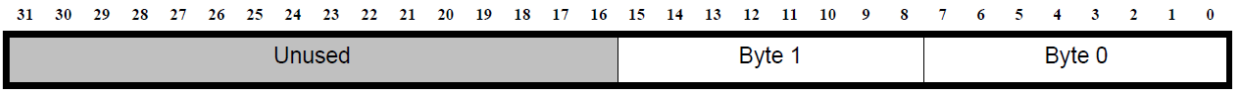

Fig. 2.9: Register Containing Sixteen-Bit Data

For the programmer's model, the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a data invariance principle.

Fig. 2.10 shows the little-endian representation of 16-bit data in byte-wide memory.

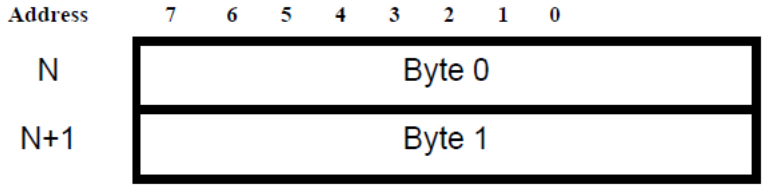

Fig. 2.10: Sixteen-Bit Register Data in Byte-Wide Memory, Little-Endian

Fig. 2.11 shows the big-endian representation.

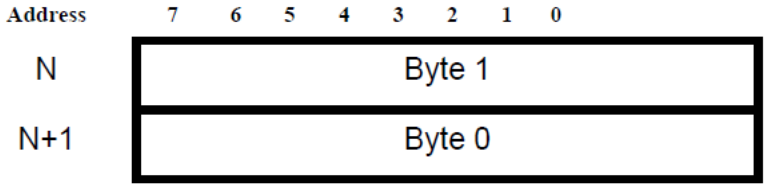

Fig. 2.11: Sixteen-Bit Register Data in Byte-Wide Memory, Big-Endian

Eight-Bit Data

Fig. 2.12 shows the 8-bit data representation in a general purpose register. For the programmer's model, the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a data invariance principle.

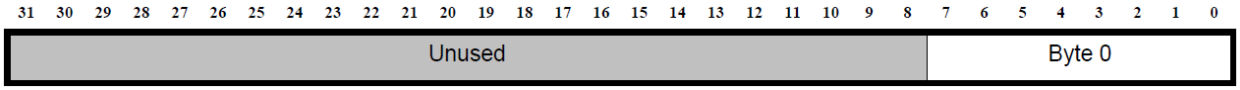

Fig. 2.12: Register Containing Eight-Bit Data

Fig. 2.13 shows the representation of 8-bit data in byte-wide memory. Regardless of the endianness of the ARCv2-based system, the byte-aligned address, n, of the byte is explicitly given and the byte is stored or read from that explicit address.

One-Bit Data

The ARCv2 instruction-set architecture supports single-bit operations on data stored in the core registers. A bit manipulation instruction includes an immediate value specifying the bit to operate on. Bit manipulation instructions can operate on 8-bit, 16-bit, or 32-bit data located within core registers because each bit is individually addressable.

Fundamental Types

Table 2.1 shows how ANSI C scalar types correspond to those of ARCv2-based processors. 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. To align the variables more strictly than what is shown in Table 2.1, fields in a struct must follow the alignment specified to ensure consistent struct mapping.

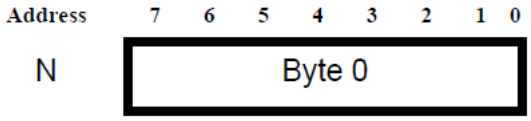

Fig. 2.13: Eight-Bit Register Data in Byte-Wide Memory

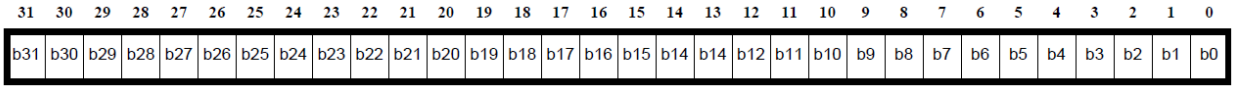

Fig. 2.14: Register Containing One-Bit Data

Table 2.1: Scalar Types History

Type	**ANSI C**	**Size**	**Alignment** (bytes)	**ARCV2 based processors**
Integral	`char` `unsigned char`	1	1	`unsigned byte`
	`signed char`	1	1	`signed byte`
	`short` `signed short`	2	2	`signed halfword`
	`unsigned short`	2	2	`unsigned halfword`

	`int`
	`signed int`
	`long`
	`signed long`
		4	4	`signed word`

	`unsigned int` `unsigned long`	4	4	`unsigned word`
	`long long` `signed long long`	8	4	`signed doubleword`

Continued on next page

Table 2.1 – continued from previous page

Type	ANSI C	Size	Alignment (bytes)	ARCV2 based processors
	`unsigned long long`	8	4	`unsigned doubleword`
Pointer	`any *` `any (*) *`	4	4	`unsigned word`
Floating	`float`	4	4	`single precision`
	`double` `long double`	8	4	`double precision`

Enumerations

The enum data type mapping is similar to that of an integer of equivalent size. Signed integral types are used by default.

Complex Types

When passed as arguments, complex data types are 32-bit word-aligned.

Aggregates and Unions

Aggregates (structures, classes, 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, classes, and unions, is always a multiple of the alignment of the object. Non-bitfield members always start on byte boundaries. The size of a struct or class is the sum of the sizes of its members, including alignment padding between members. The size of a union is the size of its largest member, padded such that its size is evenly divisible by its alignment. Enumerations can be mapped to one, two, or four bytes, depending on their size. An array uses the same alignment as its elements. Structure and union objects can be packed or padded to meet size and alignment constraints:

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member, though a packed structure or union need not be aligned on word boundaries.
- Each member is assigned to the lowest available offset with the appropriate alignment. Such alignment might 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. Such alignment might require tail padding, depending on the last member.

For detailed information on C++ classes, see "Storage Mapping for Class Objects " see *Storage Mapping for Class Objects*

In the following examples, members' byte offsets appear in the upper right corners.

Structure smaller than a word:

```
struct {
  char c;
};
```

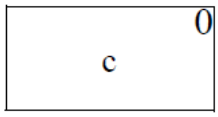

Fig. 2.15: Byte-Aligned, Sizeof is 1

No Padding:

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

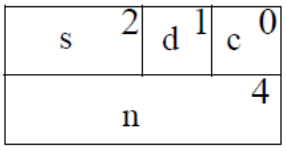

Fig. 2.16: Word-Aligned, Sizeof is 8

Internal Padding:

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

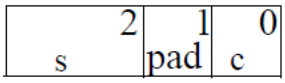

Fig. 2.17: Halfword-Aligned, Sizeof is 4

Internal and Tail Padding:

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

Union Allocation:

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

2.1.3 Storage Mapping for Class Objects

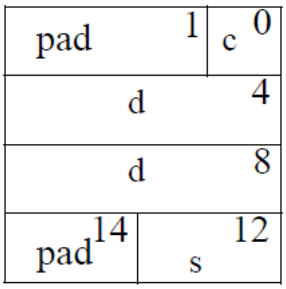

Fig. 2.18: Word-Aligned, Sizeof is 16

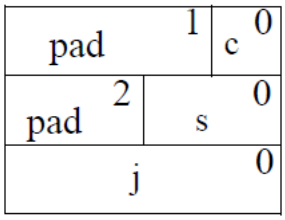

Fig. 2.19: Word-Aligned, Sizeof is 4

C++ class objects must be mapped in accordance with the GNU Itanium ABI; see the following URL: http://mentorembedded.github.io/cxx-abi/abi.html

2.1.4 Bitfields

C/C++ struct and union definitions can have bitfields, defining integral objects with a specified number of bits.

Bitfields are signed unless explicitly declared as unsigned. For example, a four-bit field declared as int can hold values from -8 to 7.

Table 2.2 shows the possible widths for bitfields, where w is maximum width (in bits).

Table 2.2: Bitfield Types-Width and Range of Values

Bit Field Type	Max Width *w* (Bits)	Range of Values
`signed char`	1 to 8	$2^{(w-1)} - 1$ to $-2^{(w-1)}$
`char` (default signedness)	1 to 8	0 to $2^w - 1$
`unsigned char`	1 to 8	0 to $2^w - 1$
`short`	1 to 16	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
`unsigned short`	1 to 16	0 to $2^w - 1$
`int`	1 to 32	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
`long`	1 to 32	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
`enum` (unless signed values are assigned)	1 to 32	0 to $2^w - 1$
`unsigned int`	1 to 32	0 to $2^w - 1$
`unsigned long`	1 to 32	0 to $2^w - 1$
`long long int`	1 to 64	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
`unsigned long long int`	1 to 64	0 to $2^w - 1$

Bitfields obey the same size and alignment rules as other structure and union members, with the following additions:

- Bitfields are allocated from most to least significant bit on big-endian implementations.

- Bitfields are allocated from least to most significant bit on little-endian implementations.
- The alignment that a bit field imposes on its enclosing struct or union is the same as any ordinary (non-bit) field of the same type. Thus, a bitfield of type int imposes a four-byte alignment on the enclosing struct.
- Bitfields are packed in consecutive bytes, except if a bitfield packed in consecutive bytes would cross a byte offset *B* where `B % sizeof(FieldType) == 0`.

  In particular:

  – A bitfield of type `char` must not cross a byte boundary.

  – A bitfield of type `short` must not cross a halfword boundary.

  – A bit field of type `int` must not cross a word boundary.

  – Because long long ints are four-byte-aligned on ARCv2-based processors, a bitfield of type `long long` must not cross two word boundaries. Thus, field B in the following code would start on byte 4 of the parent struct: :code: *struct S { int A:8; long long B:60; }*

You can insert padding as needed to comply with these rules.

Unnamed bitfields of non-zero length do not affect the external alignment. In all other respects, they behave the same as named bitfields. An unnamed bitfield of zero length causes alignment to occur at the next unit boundary, based on its type.

The struct in the following example can be mapped as illustrated in or Fig. 2.20.

```
struct {
   unsigned x:11, y:9, :0, w:13, z:1;
   char  c;
   short i;
   }
```

The *struct* in Fig. 2.20 is aligned on address boundaries divisible by four because it contains `int` types. Note that the unnamed bitfield (:0) forces padding, while alignment rules sometimes pad.

If w were changed to a *char* type, it would still be forced to begin in byte four. If there were no unnamed bitfield, w would begin in byte two, three, or four, depending on whether it could fit in the space remaining without crossing its storage-unit boundary (which is four).

The following examples show the byte offsets of *struct* and *union* members in the upper right corners for little-endian implementations. Bit numbers appear in the lower corners.

Bit numbering of `0x01020304`:

Bit-Field Allocation:

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

Boundary Alignment:

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

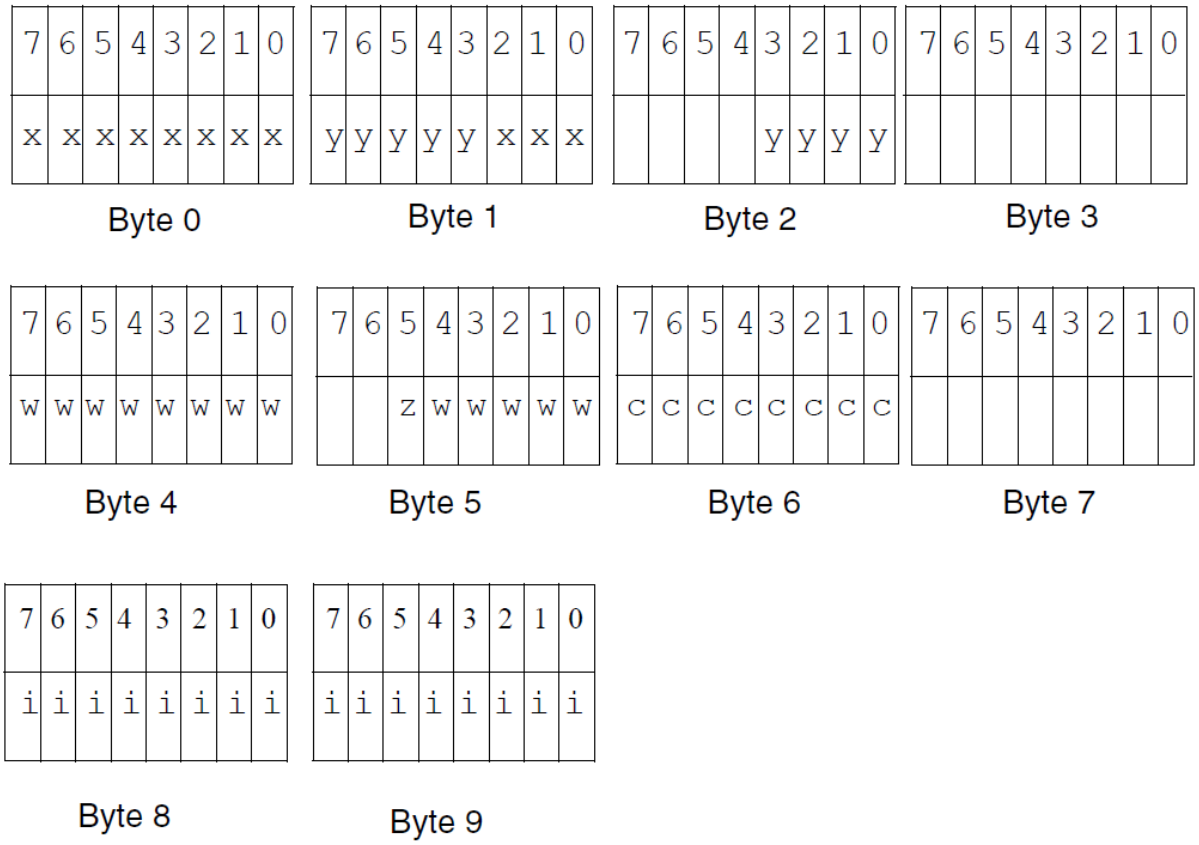

Fig. 2.20: Structure Mapped in Little-Endian Orientation

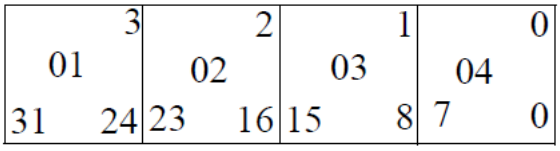

Fig. 2.21: Bit Numbering

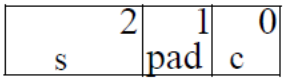

Fig. 2.22: Word-Aligned, Sizeof is 4

```
  char  d;
};
```

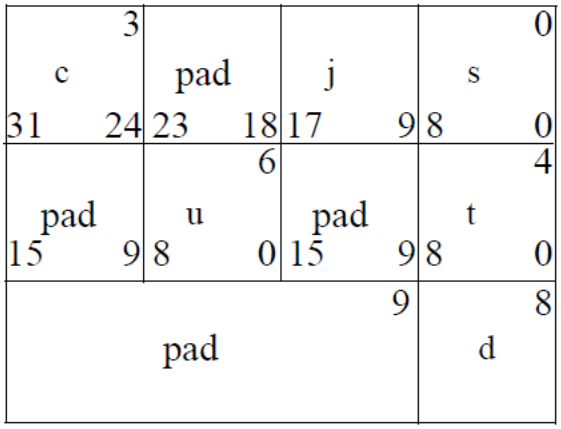

Fig. 2.23: Word-Aligned, Sizeof is 12

Storage Unit Sharing:

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

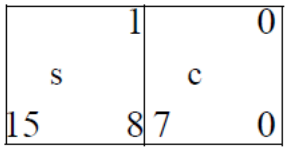

Fig. 2.24: Halfword-Aligned, Sizeof is 2

Union Allocation:

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

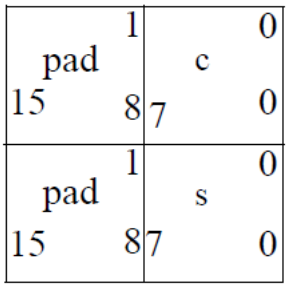

Fig. 2.25: Halfword-Aligned, Sizeof is 2

Unnamed Bitfields:

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

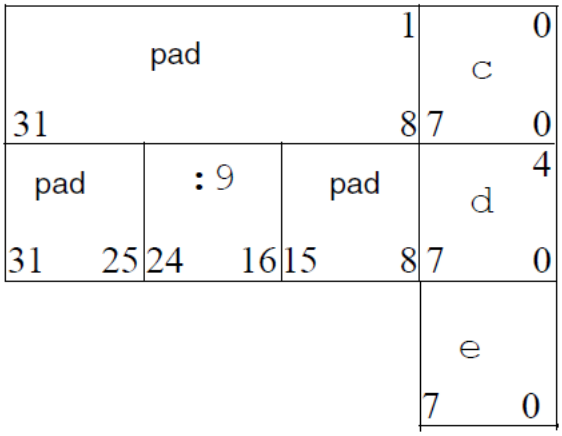

Fig. 2.26: Byte-Aligned, Sizeof is 9

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 might not be aligned in memory on suitable boundaries. For example, the d members in an array of these structures are not all on an int (four-byte) boundary. Because there is no named field with any alignment requirements beyond a byte, the struct is nine bytes wide, one-byte aligned.

2.2 Function Calling Sequence

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

Programs must follow the conventions given here. For examples of approaches permissible within these conventions, see *Coding Examples*.

2.2.1 Registers

The base-case processor hardware provides 32 word-sized (32-bit) registers and a number of special-purpose auxiliary registers. Auxiliary registers are used only by the LR and SR assembly instructions.

Core Registers

Table 2.3 and Table 2.4 summarize the registers and their functions in a standard processor build. If a reduced register set is specified, only four words of arguments are passed in registers: r0 through r3. In addition, registers r4-r9 and r16-r25 are not available with the reduced register set.

Note: Alternatively, a compiler can be configured to pass 64-bit arguments only in even/odd register pairs. In the example F(int a, long long b); argument a can be passed in r0, and argument b can be passed in r2 and r3. Note that r1 is skipped so that the 64-bit value can reside in an even/odd pair if preceded by a single 32-bit word.

Code generated with such an argument-passing mechanism is not compatible with code emitted using the mechanism described in Table 2.3. See your compiler documentation for compatibility options.

Table 2.3: General and Program-Counter Register Functions

Register	Primary Function	Secondary Function
r0	Integer result, Argument 1	Caller-saved scratch register
r1	Argument 2	Caller-saved scratch register
r2	Argument 3	Caller-saved scratch register
r3	Argument 4	Caller-saved scratch register
r4	Argument 5	Caller-saved scratch register
r5	Argument 6	Caller-saved scratch register
r6	Argument 7	Caller-saved scratch register
r7	Argument 8	Caller-saved scratch register
r8 – r12	Caller-saved scratch register	—
r13 – r24	Callee-saved register variable	—
r25	Caller-saved scratch register	gcc reserves r25 as Thread pointer if Thread local storage is enabled
r26	Small-data base register (gp)	—
r27	Frame pointer (fp)	—
r28	Stack top pointer (sp)	—
r29	Interrupt link register	—
r30	General purpose register	Caller-saved scratch register
r31	Branch link register (blink)	—
r58	Accumulator Low ACCL (little endian), ACCH (big-endian)	Caller-saved scratch register
r59	Accumulator High ACCH (little endian), ACCL (big-endian)	Caller-saved scratch register
r60	Loop counter (lp_count)	Caller-saved scratch register (compilers only—not user code)
r62	Long immediate data indicator	—
r63	program-counter value (pcl)	

The pcl register (r63) contains the four-byte-aligned value of the program counter.

The lp_count register (r60) is the 32-bit loop-counter register. It is not preserved across function calls, but you can change this behavior by including it in the registers specified with option `-Hirq_ctrl_saved="regs"` or pragma `irq_ctrl_saved("regs")`.

Note: The scratch registers are not preserved across function calls. When calling an external function, the compiler assumes that registers r0 through r12 and r30 are trashed; and that r13 through r29 are preserved. The EV6x processor reserves r25.

Auxiliary Registers

Table 2.4 summarizes the most commonly used auxiliary registers. Due to the large number of auxiliary registers possible on an ARC processor, this listing is necessarily incomplete, and might vary from one implementation to

another. See the *Programmer's Reference Manual* for a specific ARCv2-based processor for a complete listing of the auxiliary registers that can be implemented on that processor.

Table 2.4: Auxiliary-Register Functions

Address	Function
0x2	Loop start address (lp_start)
0x3	Loop end address (lp_end)
0x4	Processor identification
0x5	debug
0x6	Program counter (nextpc)
0xa	Condition flags (status32)
0xb	Status save register for highest-priority interrupt (status32_p0)
0xc	Unused
0x21	Processor-timer-0 count value
0x22	Processor-timer-0 control value
0x23	Processor-timer-0 limit value
0x25	Interrupt-vector base address
0x68	Default vector-base build configuration
0x100	Processor-timer-1 count value
0x101	Processor-timer-1 control value
0x102	Processor-timer-1 limit value
0x201	Software interrupt
0x290	JLI table base register
0x291	LDI table base register
0x292	EI table base register
0x400	Exception return address
0x401	Exception-return branch-target address
0x402	Exception-return status
0x403	Exception cause
0x404	Exception-fault address
0x410	User-mode extension enables
0x412	Branch-target address
0x413	Unused
0x414	Unused

The nextpc auxiliary register contains the program counter; the pcl register contains the 4-byte aligned value of the program counter. The status32 auxiliary register contains the condition flags.

For information on which registers can be used by which 16-bit instructions, see the *Programmer's Reference Manual* for each processor.

2.2.2 Stack Frame

This section describes the layout of the stack frame and registers that must be saved by the callee prolog code.

The Stack-Pointer Register

The stack-pointer (sp) register always points to the lowest used address of the most recently allocated stack frame. The value of sp is a four-byte-aligned address.

The stack-pointer register is commonly used as a base register to access stack-frame-based variables, which always have a positive offset. However, when alloca() is called, the stack-pointer register might be arbitrarily decremented after the stack frame is allocated. In such a case, the frame pointer register is used to reference stack-frame-based variables.

The Frame-Pointer Register

The frame pointer register (fp) is used when a function calls alloca() to allocate space on the stack, and stack-frame-based variables must be accessed.

The Callee's Prolog Code

The callee's prolog code saves all registers that need to be saved. Saved values include the value of the caller's blink (return address) register, callee-saved registers used by the function, and the frame-pointer register, if required.

The caller's stack-pointer (sp) register does not need to be saved because the compiler is able to restore the stack pointer for each function to its original value (for example, by using an add instruction).

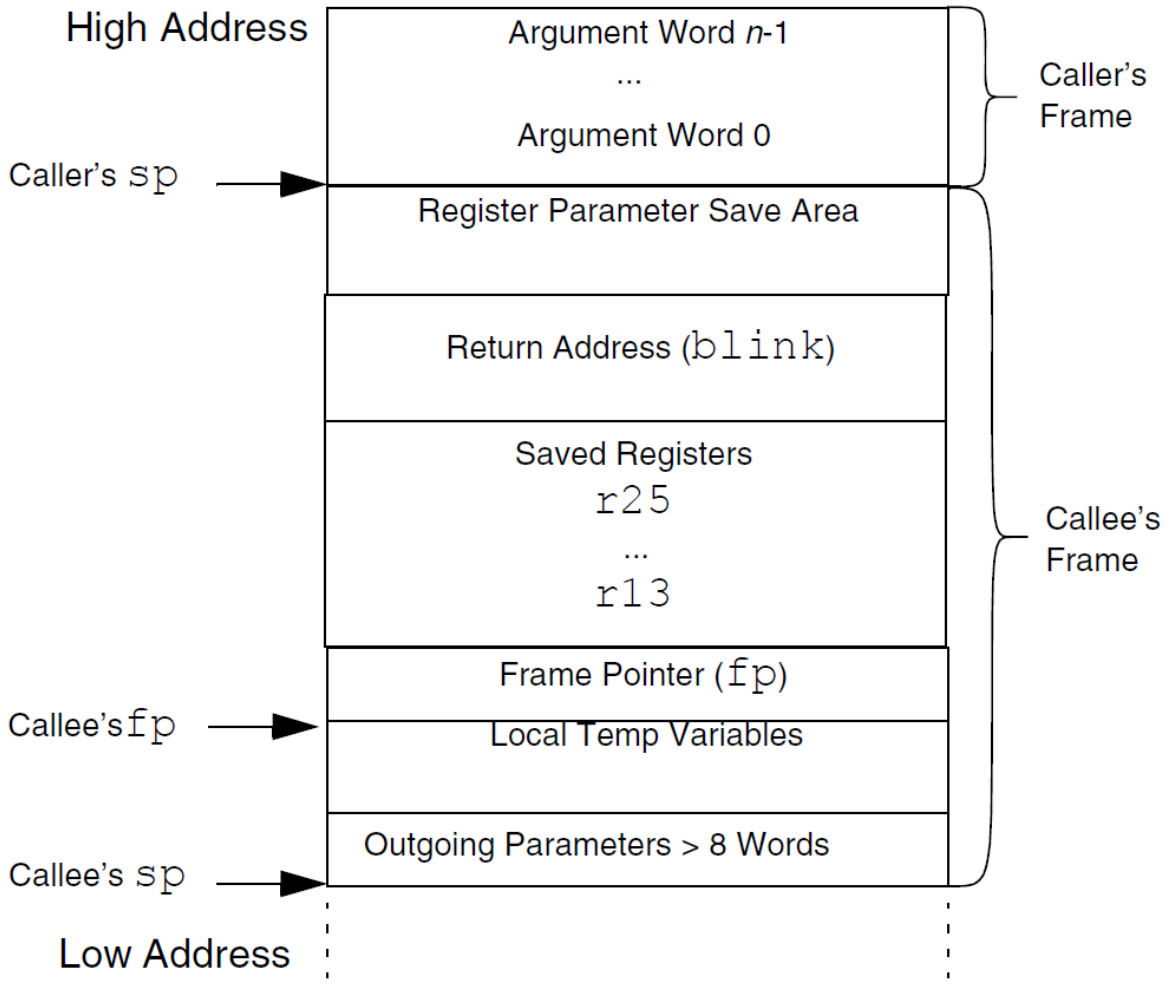

Fig. 2.27: Stack Frame for One Function Invocation

2.2.3 Allocating Stack Space Dynamically

Programs can dynamically grow the current stack frame using a memory-allocating function. The memory-allocating function must maintain a frame pointer and the stack mechanics outlined in *Stack Frame* through *Prolog and Epilog Code*. The stack frame must be maintained using the frame pointer (fp) instead of the stack pointer (sp).

2.2.4 Argument Passing

Arguments are passed as an ordered list of machine-level values from the caller to the callee.

- The first eight words (32 bytes) of arguments are loaded into registers r0 to r7. In builds with a reduced register set, the first four words are loaded into r0 to r3.
- The remaining arguments are passed by storing them into the stack immediately above the stack-pointer register.

2.2.5 Return Values

Functions return the following results:

- Any scalar or pointer type that is 32 bits or less in size (char, short, int, long) is returned in r0.
- Eight-byte integers (long long, double, and float complex) are returned in r0 and r1.
- Results of type complex double are returned in r0 to r3.
- Results of type complex float are returned in r0 and r1.
- Results of type struct are returned in a caller-supplied temporary variable whose address is passed in r0. For such functions, the arguments are shifted so that they are passed in r1 and upwards.

2.3 Process Initialization

This Supplement does not define a process-initialization state. The processor begins executing code at a hard-coded location and initially has no stack; establishing the operating environment for processes and programs entails setting up a stack and methods for passing arguments and return values as described in *Stack Frame*.

The processor supports kernel and user operating modes to permit different levels of privilege to be assigned to operating system kernels and user programs, strictly controlling access to privileged system-control instructions and special registers. Kernel mode is the default mode from reset. For more information on the operating modes, see the *Programmer's Reference Manual*.

The processor can be restarted by clearing the H bit in the STATUS32 register. On restart, the pipeline is flushed; interrupts are disabled; status register flags are cleared; the semaphore register is cleared; loop count, loop-start and loop-end registers are cleared; the scoreboard unit is cleared; the pending-load flag is cleared; and program execution resumes from the 32-bit address specified by the user as the first 32-bit entry in the interrupt-vector table, the reset vector. The core registers are not initialized except lp_count (which is cleared). A jump to the reset vector (a soft reset) does not pre-set any of the internal states of the processor. The reset value of the vector base register determines the reset vector address.

Note: User extensions and optimizations to this area are permitted.

2.4 Operating System Interface

2.4.1 Linux

OS ABI consists of system calls provided by Linux kernel and call upon by user space library code.

- ABI is similar to a regular function call in terms of arguments passing semantics. For example, 64-bit data in register pairs.
- Up to eight arguments allowed in registers r0 to r7.

- Syscall number must be passed in register r8.
- Syscall return value is returned back in r0.
- All registers except r0 are preserved by kernel across the Syscall.

The current Linux OS ABI (v4.8 kernel onwards) is ABIv4. For information on the ABI versions, see https://github.com/foss-for-synopsys-dwc-arc-processors/linux/wiki/ARC-Linux-Syscall-ABI-Compatibility

2.5 Coding Examples

This section discusses example code sequences for basic operations.

2.5.1 Prolog and Epilog Code

A function's prolog and epilog code establish the environment needed by the body of the function. This Supplement does not specify any particular prolog or epilog code, but provides the following suggested guidelines and examples; the only requirements of a function prolog are that it meet the expectations of the caller and callee, particularly as regards the passing of parameters.

- The prolog establishes a stack frame, if necessary, and can save any callee-saved registers the function uses.
- The epilog generally restores registers that were saved in the prolog code, restores the previous stack frame, and returns to the caller.

In each of the prolog-code examples in this section, framesize is the size, in bytes, of the area needed for auto variables, spill temporaries, and saved registers.

Standard Prolog Code

Standard prolog code performs the following tasks, in this order:

1. Saves the return-address (blink) register on the stack.
2. Saves any callee-saved registers that are modified by the function.
3. Allocates any additional space required in the frame by decrementing the stack pointer accordingly.

This is the standard prolog code:

```
; Save return address register:
   push_s   %blink
; Save registers r13, r14, r15, and so on
; (all callee-saved registers that must be saved):
   push_s   %r13
   push_s   %r14
   push_s   %r15
: Allocate remainder of frame
   sub      %sp, %sp, additional_space
```

Abbreviated Prolog and Epilog

For a leaf function (that does not call other functions), a compiler can abbreviate the prolog and epilog, as long as it conforms to the ABI for globally accessed functions.

Data Objects

The transfer of data to and from memory is accomplished using load and store instructions.

Volatile and Uncached Variables

The run-time model permits variables to be designated as volatile or uncached.

- A volatile variable is assumed to have a value that can asynchronously change, independent of the thread that is referencing the variable. Thus it is not advisable to cache the value of such variables or to attempt to optimize multiple accesses to them.
- Uncached variables are loaded and stored without using the processor's data cache. Use .ucdata section to store them separately.

Function Calls and Branching

Programs might use one of several branch, jump, and link instructions to control execution flow through direct and indirect function calls and branching. For function calling, the conditional branch-and-link instruction has a maximum branch range of +/- 1 MB, and the target address is 32-bit-aligned. The unconditional branch-and-link format has a maximum branch range of +/- 16 MB.

See *Programmer's Reference Manual* for your ARCv2-based processor for a list of instructions.

3 Object Files

3.1 ELF Header

3.1.1 Machine Information

For file identification in *e_ident*, ARCv2-based processors require the following values:

`e_ident[EI_CLASS] ELFCLASS32` For all 32-bit implementations

`e_ident[EI_DATA] ELFDATA2LSB` If execution environment is little-endian

`e_ident[EI_DATA] ELFDATA2MS` If execution environment is big-endian

Processor identification resides in the ELF header's e_machine member, and must have the value 195 (0xc3), defined as the name `EM_ARCOMPACT2`.

Tools may use e_flags to distinguish ARCv2-based processor families, where 5 identifies the ARC EM processor family, and 6 identifies the ARC HS processor family.

The high bits are used to select the Linux OSABI:

Table 3.1: Linux OSABI Selection

0x000	OSABI_ORIG	v2.6.35 kernel (sourceforge)
0x200	OSABI_V2	v3.2 kernel (sourceforge)
0x300	OSABI_V3	v3.9 kernel (sourceforge)
0x400	OSABI_V4	v24.8 kernel (sourceforge)

3.2 Special Sections

3.2.1 Special Sections: Types and Attributes

The sections listed in Table 3.2 are used by the system and have the types and attributes shown.

Table 3.2: Special Sections

Name	Type	Attributes
.arcextmap	SHT_PROGBITS	none
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.ctors	SHT_PROGBITS	SHF_ALLOC
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.fixtable	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.heap	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.initdata	SHT_PROGBITS	SHF_ALLOC
.offsetTable	SHT_PROGBITS	SHF_ALLOC + SHF_OVERLAY_OFFSET_TABLE + SHF_INCLUDE
.overlay	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR + SHF_OVERLAY + SHF_INCLUDE
.overlayMulti-Lists	SHT_PROGBITS	SHF_ALLOC + SHF_INCLUDE
.pictable	SHT_PROGBITS	SHF_ALLOC
.rodata_in_data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.sbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.sdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.stack	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINST
.tls	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.ucdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.vectors	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINST

Note: To be compliant with the ARCv2 ABI, a system must support *.tls*, *.sdata*, and *.sbss* sections, and must recognize, but may choose to ignore, .arcextmap and .stack sections.

3.2.2 Special Sections: Description

The special sections are described in Table 3.3

Special features might create additional sections. For details regarding overlay-related sections see the *Automated Overlay Manager User's Guide*.

Table 3.3: Special Section Description

Special Section	Description
.arcextmap	Debugging information relating to processor extensions
.bss	Uninitialized variables that are not const-qualified (startup code normally sets .bss to all zeros)
.ctors	Contains an array of functions that are called at startup to initialize elements such as C++ static variables
.data	Static variables (local and global)
.fixtable	Function replacement prologs
.heap	Uninitialized memory used for the heap
.initdata	Initialized variables and code (usually compressed) to be copied into place during run-time startup
.offsetTable	Overlay-offset table
.overlay	All overlays defined in the executable
.overlayMultiLists	Token lists for functions that appear in more than one overlay group
.pictable	Table for relocating pre-initialized data when generating position-independent code and data
.rodata_in_data	Read-only string constants when -Hharvard or -Hccm is specified.
.sbss	Uninitialized data, set to all zeros by startup code and directly accessible from the %gp register
.sdata	Initialized small data, directly accessible from the %gp register, and small uninitialized variables
.stack	Stack information
.text	Executable code
.tls	Thread-local data
.ucdata	Holds data accessed using cache bypass
.vectors	Interrupt vector table

Note: *.tls* is not necessarily the same as the *.tdata* section found in other architectures. It does not need special treatment except to be recognized as a valid .data section. It may or may not map into any current or future system thread architecture. It must remain programmable by the RTOS and application programmer as defined by the ARC MetaWare run time so that true lightweight threads can be implemented.

Caution: Sections that contribute to a loadable program segment must not contain overlapping virtual addresses.

3.3 Symbol Table

3.3.1 Symbol Values

ARCv2-based processors that support the Linux operating system follow the Linux conventions for dynamic linking.

3.4 Small-Data Area

Programs may use a small-data area to reduce code size by storing small variables in the .sdata and .sbss sections, where such data can be addressed using small, signed offsets from the %gp register. If the program uses small data,

program startup must initialize the %gp register to the address of symbol _SDA_BASE_ Such initialization is typically performed by the default startup code.

3.5 Register Information

The names and functions of the processor registers are described in *Registers*. Compilers may map variables to a register or registers as needed in accordance with the rules described in *Argument Passing* and *Return Values*, including mapping multiple variables to a single register.

Compilers may place auto variables that are not mapped into registers at fixed offsets within the function's stack frame as required, for example to obtain the variable's address or if the variable is of an aggregate type.

3.6 Relocation

3.6.1 Relocation Types

Relocation entries describe how to alter the instruction and data relocation fields shown in *Relocatable Fields*. Bit numbers appear in the lower box corners. Little-endian byte numbers appear in the upper right box corners.

3.6.2 Relocatable Fields

This document specifies several types of relocatable fields used by relocations.

bits8

Specifies 8 bits of data in a separate byte.

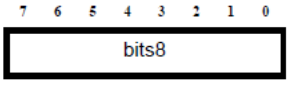

Fig. 3.1: bits8 Relocatable Field

bits16

Specifies 16 bits of data in a separate byte.

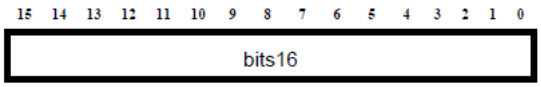

Fig. 3.2: bits16 Relocatable Field

bits24

Specifies 24 bits of data in a separate byte.

disp7u

The gray areas in Fig. 3.4 represent a disp7u relocatable field, which specifies a seven-bit unsigned displacement within a 16-bit instruction word, with bits 2-0 of the instruction stored in bits 2-0 and bits 6-3 of the instruction stored in bits 7-4.

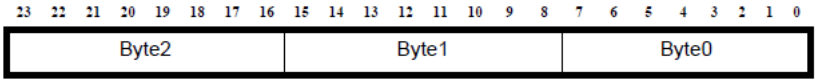

Fig. 3.3: bits24 Relocatable Field

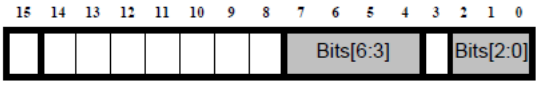

Fig. 3.4: disp7u Relocatable Field

disp9

The gray area in Fig. 3.5 represents a disp9 relocatable field, which specifies a nine-bit signed displacement within a 32-bit instruction word.

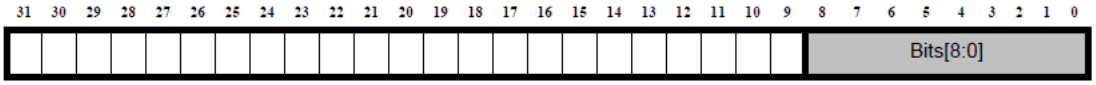

Fig. 3.5: disp9 Relocatable Field

disp9ls

The gray areas in Fig. 3.6 represent a disp9ls relocatable field, which specifies a nine-bit signed displacement within a 32-bit instruction word.

disp9s

The gray area in Fig. 3.7 represents a disp9s relocatable field, which specifies a 9-bit signed displacement within a 16-bit instruction word.

disp10u

The gray area in Fig. 3.8 represents a disp10u relocatable field, which specifies a 10-bit unsigned displacement within a 16-bit instruction word.

disp13s

The gray area in Fig. 3.9 represents a disp13s relocatable field, which specifies a signed 13-bit displacement within a 16-bit instruction word. The displacement is to a 32-bit-aligned location and thus bits 0 and 1 of the displacement are not explicitly stored.

disp21h

The gray areas in Fig. 3.10 represent a disp21h relocatable field, which specifies a 21-bit signed displacement within a 32-bit instruction word. The displacement is to a halfword-aligned target location, and thus bit 0 of the displacement is not explicitly stored. Note that the 32-bit instruction containing this relocation field may be either 16-bit-aligned or 32-bit-aligned.

disp21w

The gray areas in Fig. 3.11 represent a disp21w relocatable field, which specifies a signed 21-bit displacement within a 32-bit instruction word. The displacement is to a 32-bit-aligned target location, and thus bits 0 and 1 of the dis-

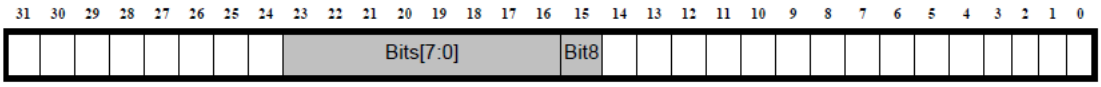

Fig. 3.6: disp9ls Relocatable Field

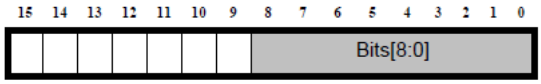

Fig. 3.7: disp9s Relocatable Field

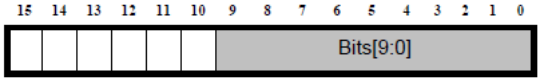

Fig. 3.8: disp10u Relocatable Field

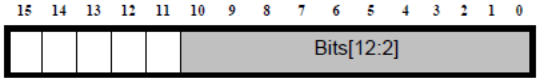

Fig. 3.9: disp13s Relocatable Field

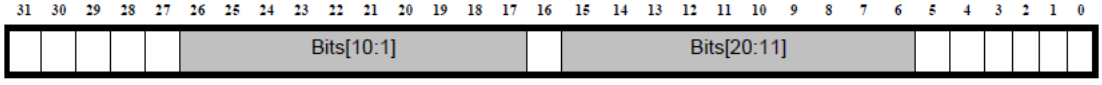

Fig. 3.10: disp21h Relocatable Field

placement are not explicitly stored. Note that the 32-bit instruction containing this relocation field may be either 16-bit-aligned or 32-bit-aligned.

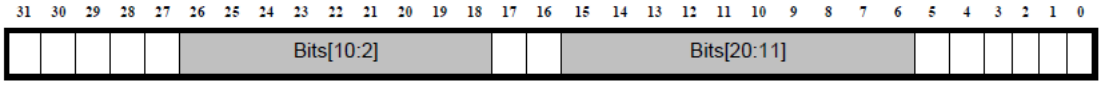

Fig. 3.11: disp21w Relocatable Field

disp25h

The gray areas in Fig. 3.12 represent a disp25h relocatable field, which specifies a 25-bit signed displacement within a 32-bit instruction word. The displacement is to a halfword-aligned target location, and thus bit 0 is not explicitly stored. Note that the 32-bit instruction containing this relocation field may be either 16-bit-aligned or 32-bit-aligned.

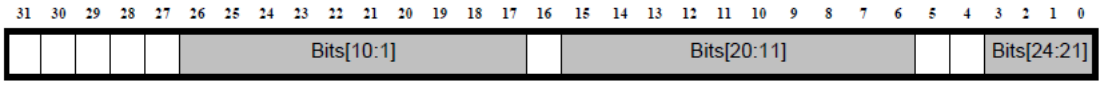

Fig. 3.12: disp25h Relocatable Field

disp25w

The gray areas in Fig. 3.13 represent a disp25w relocatable field, which specifies a 25-bit signed displacement within a 32-bit instruction word. The displacement is to a 32-bit-aligned target location, and thus bits 0 and 1 are not explicitly stored. Note that the 32-bit instruction containing this relocation field may be either 16-bit-aligned or 32-bit-aligned.

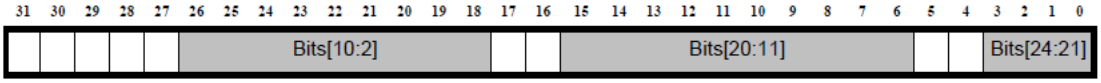

Fig. 3.13: disp25w Relocatable Field

disps9

The gray area in Fig. 3.14 represents a disps9 relocatable field, which specifies a nine-bit signed displacement within a 16-bit instruction word. The displacement is to a 32-bit-aligned location, and thus bits 0 and 1 of the displacement are not explicitly stored. This means that effectively the field is bits 10-2, stored at 8-0.

disps12

The gray areas in Fig. 3.15 represent a disps12 relocatable field, which specifies a twelve-bit signed displacement within a 32-bit instruction word. The high six bits are in 0-5, and the low six bits are in 6-11.

word32

Fig. 3.16 specifies a 32-bit field occupying four bytes, the alignment of which is four bytes unless otherwise specified. See also Fig. 3.17 and Fig. 3.18.

word32me (Little-Endian)

Specifies a 32-bit field in middle-endian Storage. Bits 31..16 are stored first, and bits 15..0 are stored adjacently. The individual halfwords are stored in the native endian orientation of the machine (little endian in Fig. 3.17).

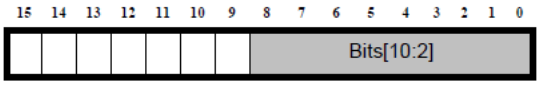

Fig. 3.14: disps9 Relocatable Field

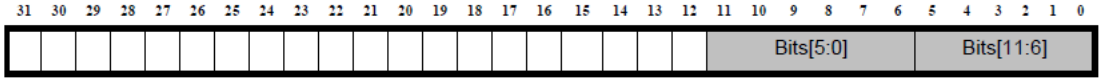

Fig. 3.15: disps12 Relocatable Field

word32me (Big-Endian)

Specifies a 32-bit field in middle-endian Storage. Bits 31..16 are stored first, and bits 15..0 are stored adjacently. The individual halfwords are stored in the native endian orientation of the machine (big endian in Fig. 3.18).

3.6.3 Relocatable-Field Calculations

The calculations presented in this section assume that the actions transform 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.

The procedure is as follows:

- Decide how to combine and locate the input files.
- Update the symbol values.
- Perform the relocation.

Relocations applied to executable or shared object files are similar and accomplish the same result.

The descriptions in this section use the following notation:

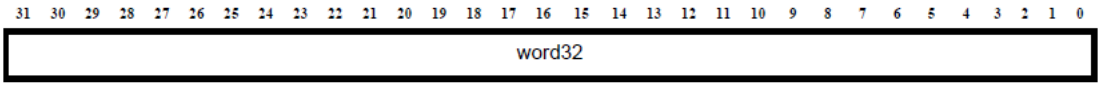

Fig. 3.16: disps12 Relocatable Field

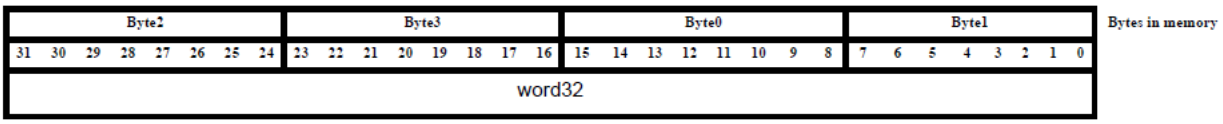

Fig. 3.17: word32me Relocatable Field on a Little-Endian Machine

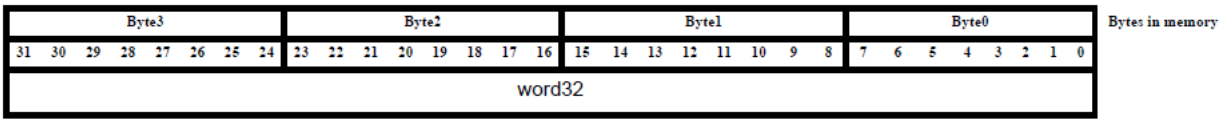

Fig. 3.18: word32me Relocatable Field on a Big-Endian Machine

Table 3.4: Relocation Terminology

Address	Function
A	The addend used to compute the value of the relocatable field
B	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.
G	The offset into the global offset table at which the address of the relocated symbol will reside during execution.
GOT	The address of the global offset table
L	The place (section offset or address) of the PLT 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 associated GOT entries during execution.
MES	Middle-Endian Storage A 32-bit word is stored in two halfwords, with bits 31..16 stored first and bits 15..0 stored adjacently. The individual halfwords are stored in the native endian orientation of the machine. This type of storage is used for all instructions and long immediate operands in the ARCv2 architecture.
P	The place (section offset or address) of the storage unit being relocated (computed using r_offset)
S	The value of the symbol whose index resides in the relocation entry
SECTSTART	Start of the current section. Used in calculating offset types.
_SDA_BASE_	Base of the small-data area
JLI	Base of the JLI table

A relocation entry's r_offset value designates the offset or virtual address of the first byte of the field to be relocated. The relocation type specifies which bits to change and how to calculate their values. The ARCv2 architecture uses only Elf32_Rela relocation entries. The addend is contained in the relocation entry. Any data from the field to be relocated is discarded. In all cases, the addend and the computed result use the same byte order.

Note: With the exception of word32, all relocations with replacement fields in four-byte words must be written using Middle-Endian Storage.

Table 3.5: Relocation Types

Name	Value	Field	Calculation
R_ARC_NONE	0x0	none	none
R_ARC_8	0x1	bits8	S+A
R_ARC_16	0x2	bits16	S+A
R_ARC_24	0x3	bits24	S+A
R_ARC_32	0x4	word32	S+A
R_ARC_N8	0x8	bits8	A–S
R_ARC_N16	0x9	bits16	A–S
R_ARC_N24	0xa	bits24	A–S
R_ARC_N32	0xb	word32	P - (S+A)
R_ARC_SDA	0xc	disp9	S–_SDA_BASE_ +A
R_ARC_SECTOFF	0xd	word32	(S-SECTSTART)+A
R_ARC_S21H_PCREL	0xe	disp21h	(S+A-P)>>1 (convert to halfword displacement)
R_ARC_S21W_PCREL	0xf	disp21w	(S+A-P)>>2 (convert to longword dis+D69placement)
R_ARC_S25H_PCREL	0x10	disp25h	(S+A-P)>>1 (convert to halfword displacement)
R_ARC_S25W_PCREL	0x11	disp25w	(S+A-P)>>2 (convert to longword displacement)
R_ARC_SDA32	0x12	word32	(S+A)-_SDA_BASE_
R_ARC_SDA_LDST	0x13	disp9ls	(S+A-_SDA_BASE_) (s9 range)
R_ARC_SDA_LDST1	0x14	disp9ls	(S+A-_SDA_BASE_) >>1 (s10 range)
R_ARC_SDA_LDST2	0x15	disp9ls	(S+A-_SDA_BASE_) >>2 (s11 range)
R_ARC_SDA16_LD	0x16	disp9s	(S+A-_SDA_BASE_) (s9 range)
R_ARC_SDA16_LD1	0x17	disp9s	(S+A-_SDA_BASE_) >>1 (s10 range)
R_ARC_SDA16_LD2	0x18	disp9s	(S+A-_SDA_BASE_) >>2 (s11 range)
R_ARC_S13_PCREL	0x19	disp13s	(S+A-P) >>2
R_ARC_W	0x1a	word32	(S+A) & ~3 (word-align)
R_ARC_32_ME	0x1b	word32me	S+A (MES)
R_ARC_N32_ME	0x1c	word32me	(ME (A-S))
R_ARC_SECTOFF_ME	0x1d	word32me	(S-SECTSTART)+A (MES)
R_ARC_SDA32_ME	0x1e	word32me	(S+A)-_SDA_BASE_ (MES)
R_ARC_W_ME	0x1f	word32me	(S+A) & ~3 (word-aligned MES)
R_AC_SECTOFF_U8	0x23	disp9ls	S+A-SECTSTART
R_AC_SECTOFF_U8_1	0x24	disp9ls	(S+A-SECTSTART) >>1
R_AC_SECTOFF_U8_2	0x25	disp9ls	(S+A-SECTSTART) >>2
R_AC_SECTOFF_S9	0x26	disp9ls	S+A-SECTSTART - 256
R_AC_SECTOFF_S9_1	0x27	disp9ls	(S+A-SECTSTART - 256) >>1
R_AC_SECTOFF_S9_2	0x28	disp9ls	(S+A-SECTSTART - 256) >>2
R_ARC_SECTOFF_ME_1	0x29	word32me	((S-SECTSTART)+A) >>1 (MES)
R_ARC_SECTOFF_ME_2	0x2a	word32me	((S-SECTSTART)+A) >>2 (MES)
R_ARC_SECTOFF_1	0x2b	word32	((S-SECTSTART)+A) >>1
R_ARC_SECTOFF_2	0x2c	word32	((S-SECTSTART)+A) >>2
R_ARC_SDA_12	0x2d	disps12	(S + A) - _SDA_BASE_
R_ARC_LDI_SECTOFF1	0x2e	disp7u	(S - <ldi-table base> + A) >> 2
R_ARC_LDI_SECTOFF2	0x2f	disps12	(S - <ldi-table base> + A) >> 2
R_ARC_SDA16_ST2	0x30	disps9	(S+A-_SDA_BASE) >> 2
R_ARC_PC32	0x32	word32	S+A-P
R_ARC_GOTPC32	0x33	word32	GOT+G+A-P
R_ARC_PLT32	0x34	word32	L+A-P
R_ARC_COPY	0x35	none	none
R_ARC_GLOB_DAT	0x36	word32	S
R_ARC_JMP_SLOT	0x37	word32	S

Continued on next page

Table 3.5 – continued from previous page

Name	Value	Field	Calculation
R_ARC_RELATIVE	0x38	word32	B+A
R_ARC_GOTOFF	0x39	word32	S+A-GOT
R_ARC_GOTPC	0x3a	word32	GOT+A-P
R_ARC_GOT32	0x3b	word32	G+A
R_ARC_S25H_PCREL_PLT	0x3d	disp25w	L+A-P
R_ARC_JLI_SECTOFF	0x3f	disp10u	S–JLI
R_ARC_AOM_TOKEN_ME	0x40	word32me	AOM token (32 bits)(MES)
R_ARC_AOM_TOKEN	0x41	word32	AOM token (32 bits)
R_ARC_TLS_DTPMOD	0X42		
R_ARC_TLS_DTPOFF	0X43		
R_ARC_TLS_TPOFF	0x44		
R_ARC_TLS_GD_GOT	0x45		
R_ARC_TLS_GD_LD	0X46		
R_ARC_TLS_GD_CALL	0X47		
R_ARC_TLS_IE_GOT	0X48		
R_ARC_TLS_DTPOFF_S9	0X49		
R_ARC_TLS_LE_S9	0X4A		
R_ARC_TLS_LE_32	0X4B		
R_ARC_S25W_PCREL_PLT	0x4c	disp25w	L+A-P
R_ARC_S21H_PCREL_PLT	0x4d	disp21h	L+A-P

A relocation entry's r_offset value designates the offset or virtual address of the first byte of the field to be relocated. The relocation type specifies which bits to change and how to calculate their values. The ARCv2 architecture uses only Elf32_Rela relocation entries. The addend is contained in the relocation entry. Any data from the field to be relocated is discarded. In all cases, the addend and the computed result use the same byte order.

Note: With the exception of word32, all relocations with replacement fields in four-byte words must be written using Middle-Endian Storage.

R_ARC_S21H_PCREL

This relocation type is used with conditional branches, for example:

```
bne label
```

R_ARC_S21W_PCREL

This relocation type is used with conditional branch and link, for example:

```
blne label
```

R_ARC_S25H_PCREL

This relocation type is used with unconditional branches, for example:

```
b label
```

R_ARC_S25W_PCREL

This relocation type is used with unconditional branch and link, for example:

```
bl printf
```

R_ARC_SDA32

This relocation type is used with 32-bit small-data fixups, for example:

```
add   r0, gp, var@sda
```

R_ARC_SDA_LDST*

The R_ARC_SDA_LDST* relocation types are used with small-data fixups on loads and stores. Examples:

```
ldb   r0,  [gp, var@sda]   ; R_ARC_SDA_LDST
stw   r0,  [gp, var@sda]   ; R_ARC_SDA_LDST1
ld    r0,  [gp, var@sda]   ; R_ARC_SDA_LDST2
```

R_ARC_SDA16_LD*

The R_ARC_SDA16_LD* relocation types are used with 16-bit GP-relative load instructions, when such instructions load a small-data variable relative to the GP. Examples:

```
ldb_s  r0, [gp, var@sda]   ; R_ARC_SDA16_LD
ldw_s  r0, [gp, var@sda]   ; R_ARC_SDA16_LD1
ld_s   r0, [gp, var@sda]   ; R_ARC_SDA16_LD2
```

R_ARC_S13_PCREL

This relocation type is used with 16-bit branch and link, for example:

```
bl_s printf
```

R_ARC_W

This relocation type is used to ensure 32-bit alignment of a fixup value. Examples:

```
mov   r0,  var@l
ld    r0, [pcl, lab - .@l]
```

R_ARC_*_ME

Relocation types ending in _ME behave like the non-ME relocation type of the same name, with the exception that they use Middle-Endian Storage: A 32-bit word is stored in two halfwords, with bits 31..16 stored first and bits 15..0 stored adjacently. The individual halfwords are stored in the native endian orientation of the machine. That is, the upper halfwords both have bits 31..16, but they are in a different sequence between big and little endian.

This type of storage is used for all instructions and long immediate operands in the ARCv2 architecture.

R_AC_SECTOFF*

The R_AC_SECTOFF* relocation types allow a section-relative offset for ARCv2 loads and stores in the short-immediate-operand range of 0 to 255 (-256 to -255 for the S9 variety), so long as the base register is loaded with the address of the section. Addressing may be scaled such that the range for halfwords is 0 to 510 (-256 to -510) and the range for 32-bit word accesses is 0 to 1020 (-256 to -1020), with byte accesses retaining the range 0 to 255 or -256 to -255. Examples:

```
ldb  r0, [r20, var@sectoff_u8]   ; R_AC_SECTOFF_U8
stw  r0, [r20, var@sectoff_u8]   ; R_AC_SECTOFF_U8_1
ld   r0, [r20, var@sectoff_u8]   ; R_AC_SECTOFF_U8_2
ldb  r0, [r20, var@sectoff_s9]   ; R_AC_SECTOFF_S9
stw  r0, [r20, var@sectoff_s9]   ; R_AC_SECTOFF_S9_1
ld   r0, [r20, var@sectoff_s9]   ; R_AC_SECTOFF_S9_2
```

Note: The ninth bit of the replacement field is not used for the following relocation types: - R_AC_SECTOFF_U8 - R_AC_SECTOFF_U8_1 - R_AC_SECTOFF_U8_2

R_ARC_SECTOFF*

The R_ARC_SECTOFF* relocation types are used with section-relative offset loads and stores from or to XY memory.

R_ARC_GOTPC32

This relocation type is used to obtain a PC-relative reference to the GOT entry for a symbol. This type is used for the same purpose as R_ARC_GOT32 but uses PC-relative addressing to reference the GOT whereas type R_ARC_GOT32 is typically used with a base register containing the address of the GOT. Example:

```
ld r0, [pcl, var@gotpc]
```

R_ARC_PLT32

This relocation type computes the address of the symbol's PLT entry and additionally instructs the link editor to build a procedure linkage table. This relocation type is usually not explicitly needed, as the link editor converts function calls to use this type when building a shared library or dynamic executable. Example:

```
bl func@plt
```

R_ARC_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_ARC_GLOB_DAT

The link editor creates this relocation type for dynamic linking. This relocation type is 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_ARC_JMP_SLOT

The link editor creates this relocation type for dynamic linking. Its offset member gives the location of a PLT's GOT entry. The dynamic linker modifies the GOT entry so that the PLT will transfer control to the designated symbol's address.

You might add examples after implementation is finished (mj, 1/05).

R_ARC_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_ARC_GOTOFF

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. This relocation type is not used for loading from the contents of the GOT, but to use the global data pointer anchored at the GOT to access other nearby data.

Example:

```
add r0, gp, var@gotoff
```

R_ARC_GOTPC

This relocation type resembles R_ARC_PC32, except it uses the address of the global offset table in its calculation. The symbol referenced in this relocation is _GLOBAL_OFFSET_TABLE_, which additionally instructs the link editor to build the global offset table. This relocation type provides a PC-relative means of obtaining the address of the global offset table. Example:

```
add gp, pcl, _GLOBAL_OFFSET_TABLE_@gotpc
```

3.6.4 Relocation Table

```
# Generic
#Relocation.new("R_ARC_NONE          0x0   none      bitfield    none")
Relocation.new("R_ARC_8              0x1   bits8     bitfield    S+A")
Relocation.new("R_ARC_16             0x2   bits16    bitfield    S+A")
Relocation.new("R_ARC_24             0x3   bits24    bitfield    S+A")
Relocation.new("R_ARC_32             0x4   word32    bitfield    S+A")

# Unsupported
Relocation.new("R_ARC_N8             0x8   bits8     bitfield    A-S")
Relocation.new("R_ARC_N16            0x9   bits16    bitfield    A-S")
Relocation.new("R_ARC_N24            0xa   bits24    bitfield    A-S")
Relocation.new("R_ARC_N32            0xb   word32    bitfield    A-S")
Relocation.new("R_ARC_SDA            0xc   disp9     bitfield    ME(S+A-_SDA_BASE_)")
Relocation.new("R_ARC_SECTOFF        0xd   word32    bitfield    (S-SECTSTART)+A")

# arcompact elf me reloc
Relocation.new("R_ARC_S21H_PCREL     0xe   disp21h   signed      ME((S+A-P)>>1)␣
↪(convert to halfword displacement)")
Relocation.new("R_ARC_S21W_PCREL     0xf   disp21w   signed      ME((S+A-P)>>2)␣
↪(convert to longword displacement)")
Relocation.new("R_ARC_S25H_PCREL     0x10  disp25h   signed      ME((S+A-P)>>1)␣
↪(convert to halfword displacement)")
Relocation.new("R_ARC_S25W_PCREL     0x11  disp25w   signed      ME((S+A-P)>>2)␣
↪(convert to longword displacement)")
Relocation.new("R_ARC_SDA32          0x12  word32    signed      ME((S+A)-_SDA_BASE_)")
Relocation.new("R_ARC_SDA_LDST       0x13  disp9ls   signed      ME((S+A-_SDA_BASE_))␣
↪(s9 range)")
Relocation.new("R_ARC_SDA_LDST1      0x14  disp9ls   signed      ME((S+A-_SDA_BASE_)>>
↪1) (s10 range)")
Relocation.new("R_ARC_SDA_LDST2      0x15  disp9ls   signed      ME((S+A-_SDA_BASE_)>>
↪2) (s11 range)")

# Short instructions should no be marked as ME
Relocation.new("R_ARC_SDA16_LD       0x16  disp9s    signed      (S+A-_SDA_BASE_) (s9␣
↪range)")
Relocation.new("R_ARC_SDA16_LD1      0x17  disp9s    signed      (S+A-_SDA_BASE_)>>1␣
↪(s10 range)")
Relocation.new("R_ARC_SDA16_LD2      0x18  disp9s    signed      (S+A-_SDA_BASE_)>>2␣
↪(s11 range)")
Relocation.new("R_ARC_S13_PCREL      0x19  disp13s   signed      ((S+A-P)>>2)")

# Unsupported
Relocation.new("R_ARC_W              0x1a  word32    bitfield    (S+A)&~3 (word-align)
↪")

# arcompact elf me reloc
Relocation.new("R_ARC_32_ME          0x1b  limm      signed      ME(S+A) (MES)")
```

```
# TODO: This is a test relocation
Relocation.new("R_ARC_32_ME_S       0x69  limms     signed      ME(S+A) (MES)")

# Unsupported
Relocation.new("R_ARC_N32_ME        0x1c  word32    bitfield    ME(A-S) (MES)")
Relocation.new("R_ARC_SECTOFF_ME    0x1d  word32    bitfield    ME((S-SECTSTART)+A)␣
↪(MES)")

# arcompact elf me reloc
Relocation.new("R_ARC_SDA32_ME      0x1e  limm      signed      (S+A-_SDA_BASE_)")

# Unsupported
Relocation.new("R_ARC_W_ME          0x1f  word32    bitfield    ME((S+A)&~3) (word-
↪aligned MES)")
Relocation.new("R_AC_SECTOFF_U8     0x23  disp9ls   bitfield    ME(S+A-SECTSTART)")
Relocation.new("R_AC_SECTOFF_U8_1   0x24  disp9ls   bitfield    ME((S+A-SECTSTART)>>1)
↪")
Relocation.new("R_AC_SECTOFF_U8_2   0x25  disp9ls   bitfield    ME((S+A-SECTSTART)>>2)
↪")

Relocation.new("R_AC_SECTOFF_S9     0x26  disp9ls   bitfield    ME(S+A-SECTSTART-256)
↪")
Relocation.new("R_AC_SECTOFF_S9_1   0x27  disp9ls   bitfield    ME((S+A-SECTSTART-
↪256)>>1)")
Relocation.new("R_AC_SECTOFF_S9_2   0x28  disp9ls   bitfield    ME((S+A-SECTSTART-
↪256)>>2)")


Relocation.new("R_ARC_SECTOFF_ME_1  0x29  word32    bitfield    ME(((S-SECTSTART)+A)>>
↪1) (MES)")
Relocation.new("R_ARC_SECTOFF_ME_2  0x2a  word32    bitfield    ME(((S-SECTSTART)+A)>>
↪2) (MES)")
Relocation.new("R_ARC_SECTOFF_1     0x2b  word32    bitfield    ((S-SECTSTART)+A)>>1")
Relocation.new("R_ARC_SECTOFF_2     0x2c  word32    bitfield    ((S-SECTSTART)+A)>>2")

Relocation.new("R_ARC_SDA_12        0x2d  disp12s   signed      (S+A-_SDA_BASE_)")

Relocation.new("R_ARC_SDA16_ST2     0x30  disp9s1   signed      (S+A-_SDA_BASE_)>>2␣
↪(Dsiambiguation for several relocations)")

# arcompact elf me reloc
Relocation.new("R_ARC_32_PCREL      0x31  word32    signed      S+A-PDATA")
Relocation.new("R_ARC_PC32          0x32  word32    signed      ME(S+A-P)")

# Special
Relocation.new("R_ARC_GOT32         0x3b  word32    dont        G+A") # == Special

# arcompact elf me reloc
Relocation.new("R_ARC_GOTPC32       0x33  word32    signed      ME(GOT+G+A-P)")
Relocation.new("R_ARC_PLT32         0x34  word32    signed      ME(L+A-P)")
Relocation.new("R_ARC_COPY          0x35  none      signed      none")
Relocation.new("R_ARC_GLOB_DAT      0x36  word32    signed      S")
Relocation.new("R_ARC_JMP_SLOT      0x37  word32    signed      ME(S)")
Relocation.new("R_ARC_RELATIVE      0x38  word32    signed      ME(B+A)")
Relocation.new("R_ARC_GOTOFF        0x39  word32    signed      ME(S+A-GOT)")
Relocation.new("R_ARC_GOTPC         0x3a  word32    signed      ME(GOT_BEGIN-P)")
```

```
Relocation.new("R_ARC_S21W_PCREL_PLT  0x3c  disp21w    signed      ME((L+A-P)>>2)")
Relocation.new("R_ARC_S25H_PCREL_PLT  0x3d  disp25h    signed      ME((L+A-P)>>1)")

# WITH TLS
Relocation.new("R_ARC_TLS_DTPMOD    0x42  word32    dont        0") # , 0, 2, 32,␣
↪FALSE, 0, arcompact_elf_me_reloc, "R_ARC_TLS_DTPOFF",-1),
Relocation.new("R_ARC_TLS_TPOFF     0x44  word32    dont        0") # ,"R_ARC_TLS_
↪TPOFF"),
Relocation.new("R_ARC_TLS_GD_GOT    0x45  word32    dont        ME(G+GOT-P)") # , 0,␣
↪2, 32, FALSE, 0, arcompact_elf_me_reloc, "R_ARC_TLS_GD_GOT",-1),
Relocation.new("R_ARC_TLS_GD_LD     0x46  none      dont        0") # ,"R_ARC_TLS_GD_
↪LD"),
Relocation.new("R_ARC_TLS_GD_CALL   0x47  word32    dont        0") # ,"R_ARC_TLS_GD_
↪CALL"),
Relocation.new("R_ARC_TLS_IE_GOT    0x48  word32    dont        ME(G+GOT-P)") # , 0,␣
↪2, 32, FALSE, 0, arcompact_elf_me_reloc, "R_ARC_TLS_IE_GOT",-1),
Relocation.new("R_ARC_TLS_DTPOFF    0x43  word32    dont        ME(S-SECTSTART+A)") #␣
↪, 0, 2, 32, FALSE, 0, arcompact_elf_me_reloc, "R_ARC_TLS_DTPOFF",-1),
Relocation.new("R_ARC_TLS_DTPOFF_S9 0x49  word32    dont        ME(S-SECTSTART+A)") #␣
↪, 0, 2, 32, FALSE, 0, arcompact_elf_me_reloc, "R_ARC_TLS_DTPOFF_S9",-1),
Relocation.new("R_ARC_TLS_LE_S9     0x4a  word32    dont        ME(S+TCB_SIZE-TLS_REL)
↪") # , 0, 2, 9, FALSE, 0, arcompact_elf_me_reloc, "R_ARC_TLS_LE_S9",-1),
Relocation.new("R_ARC_TLS_LE_32     0x4b  word32    dont        ME(S+A+TCB_SIZE-TLS_
↪REL)") # , 0, 2, 32, FALSE, 0, arcompact_elf_me_reloc, "R_ARC_TLS_LE_32",-1),
# WITHOUT TLS

Relocation.new("R_ARC_S25W_PCREL_PLT         0x4c  disp25w    signed      ME((L+A-P)>>
↪2)")
Relocation.new("R_ARC_S21H_PCREL_PLT         0x4d  disp21h    signed      ME((L+A-P)>>
↪1)")
```

4 Program Loading and Dynamic Linking

This section discusses loading and linking requirements for ARCv2-based processors that support UNIX-style operating systems, including Linux.

4.1 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 with offsets and virtual addresses that are congruent, modulo the page size.

Virtual addresses and file offsets for segments in ARCv2-based processors are congruent modulo 64 K bytes (0x10000) or larger powers of two. Because 64 K bytes is the maximum page size, the files are suitable for paging regardless of physical page size.

The value of the p_align member of each program header in a shared object file must be 0x10000.

Fig. 4.1 is an example of an executable file assuming an executable program linked with a base address of 0x10000000.

The following are possible corresponding program header segments:

Table 4.1: Special Section Description

Member	text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x10000100	0x1003bf00
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

Although the file offsets and virtual addresses are congruent modulo 64,000 bytes, the beginning and end file pages of each segment group can be impure. No restriction applies to the number or order of the segment groups, but ELF files traditionally contain one code group followed by one data group. For such a traditional single text and data file, up to four file pages can hold impure text or data (depending on the page size and file system block size).

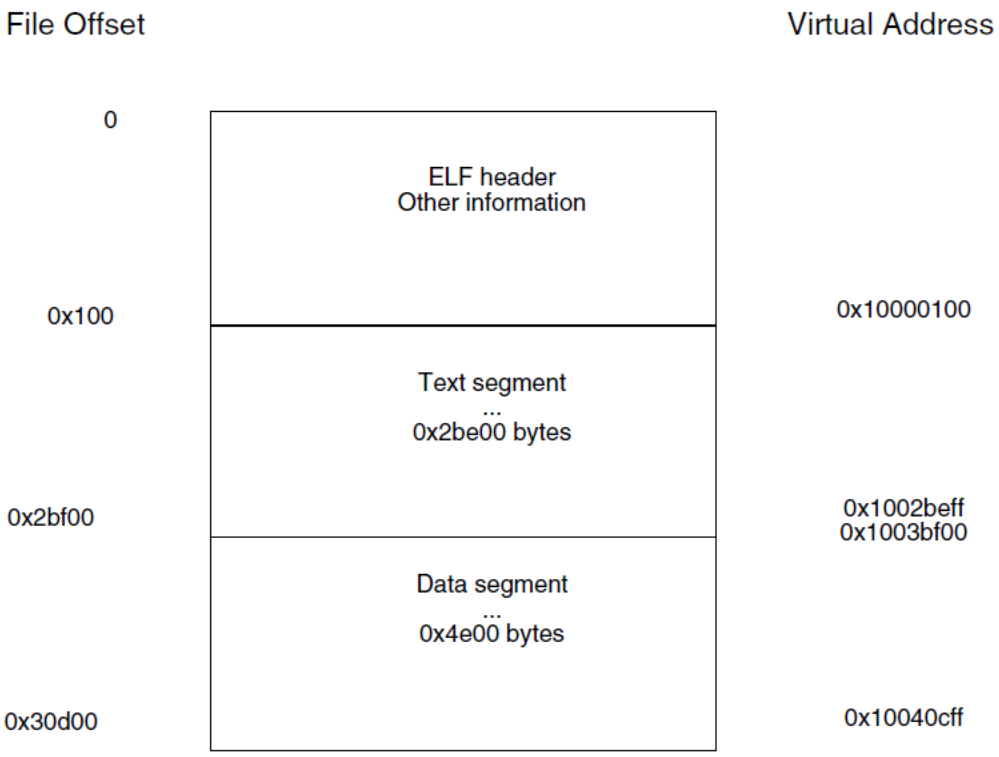

Fig. 4.1: Executable File Layout

- The first text page usually contains the ELF header, and other information.
- The last text page typically contains a copy of the beginning of data.
- The first data page usually contains a copy of the end of text.
- The last data page typically contains 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 Figure 4-1, the file region holding the end of text and the beginning of data is mapped twice; at one virtual address for text and at a different virtual address for data.

The end of a data segment requires special handling if it is followed contiguously by the uninitialized data (.bss), which is required to be initialized at startup with zeros. So if the last page of a file's representation of a data segment includes information that is not part of the segment, the extraneous data must be set to zero, rather than to the unknown contents. "Impurities" in the other start and end 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 pages with a size of 4096 (0x1000) bytes.

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.

However, shared object segments typically contain position-independent code. This allows a segment's virtual address to change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the relative positions of the segments. The difference between virtual addresses in memory must match the difference between virtual addresses in the file.

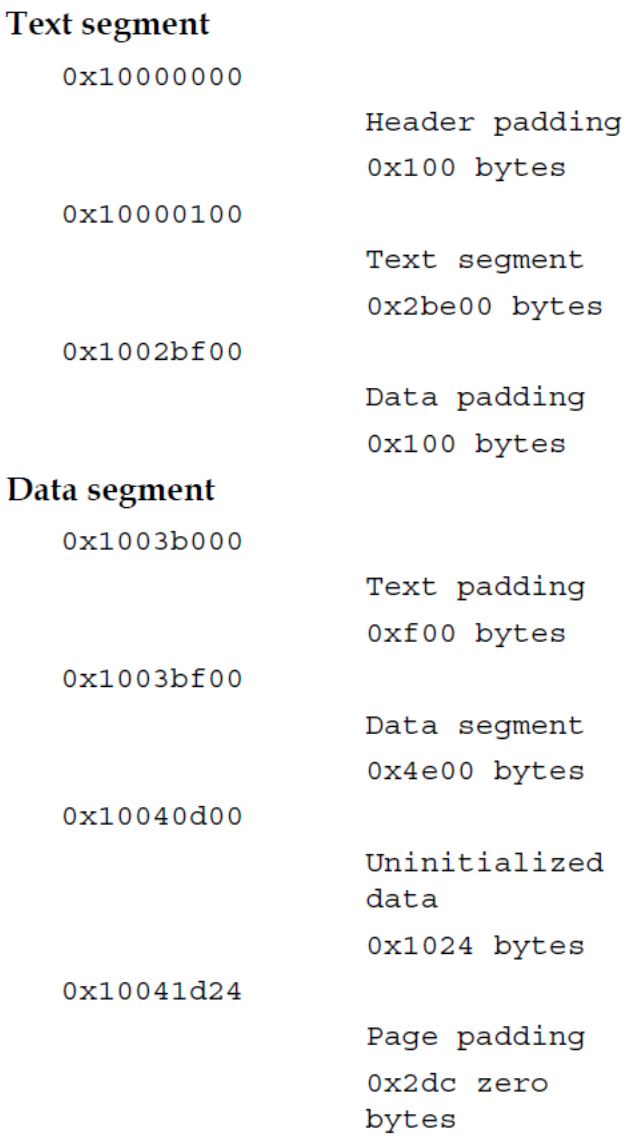

Fig. 4.2: Virtual Address

Table 4.2 shows the virtual-address assignments for shared objects that are possible for several processes, illustrating constant relative positioning. The table also illustrates the base-address computations.

Table 4.2: Virtual Address Assignments

source	text	Data	Base Address
File	0x000200	0x02a400	—
Process1	0x100200	0x12a400	0x100000
Process2	0x200200	0x22a400	0x200000
Process3	0x300200	0x32a400	0x300000
Process4	0x400200	0x42a400	0x400000
Process1	0x100200	0x12a400	0x100000

4.1.1 Program Interpreter

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

4.2 Dynamic Linking

4.2.1 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

The d_ptr member for this entry 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 ARCv2-based processors, 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 "Procedure Linkage Table " for more information.

4.2.2 Global Offset Table

Position-independent code generally may not contain absolute virtual addresses. The global offset table (GOT) holds absolute addresses in private data, making the addresses available without compromising the position independence and sharability of a program's text. A program references its GOT using position-independent addressing and extracts absolute values, 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 are of type R_ARC_GLOB_DAT, referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the GOT entries to the proper values.

Because the executable file and shared objects have separate global offset tables, a symbol might appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, ensuring the absolute addresses are available during execution.

The dynamic linker may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses after 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 .got ELF section in an executable or shared object. The symbol _GLOBAL_OFFSET_TABLE_ can be used to access the table. This symbol can reside in the middle of the .got section, allowing both positive and negative subscripts into the array of addresses.

The entry at _GLOBAL_OFFSET_TABLE_[0] is reserved for the address of the dynamic structure, referenced with the symbol _DYNAMIC. This allows the dynamic linker to find its dynamic structure prior to the processing of the relocations.

The entry at _GLOBAL_OFFSET_TABLE_[1] is reserved for use by the dynamic loader. The entry at _GLOBAL_OFFSET_TABLE_[2] is reserved to contain a dynamic the lazy symbol-resolution entry point.

If no explicit .pltgot is used, _GLOBAL_OFFSET_TABLE_[3 .. 3+F] are used for resolving function references, and _GLOBAL_OFFSET_TABLE_[3+F+1 .. last] are for resolving data references. Addressability to the global offset table (GOT) is accomplished using direct PC-relative addressing. There is no need for a function to materialize an explicit base pointer to access the GOT. GOT-based variables can be referenced directly using a single eight-byte long-intermediate-operand instruction:

```
ld rdest,[pcl,varname@gotpc]
```

Similarly, the address of the GOT can be computed relative to the PC:

```
add rdest,pcl, _GLOBAL_OFFSET_TABLE_ @gotpc
```

This add instruction relies on the universal placement of the address of the _DYNAMIC variable at location 0 of the GOT.

4.2.3 Function Addresses

References to the address of a function from an executable file or shared object and the shared objects associated with it might not resolve to the same value. References from within shared objects are normally 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 are normally 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 places the address of the PLT entry for that function in the associated symbol-table entry. 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.

If the st_shndx member of the symbol-table entry is not SHN_UNDEF, the dynamic linker has found a definition for the symbol and uses its st_value member as the symbol's address. If the st_shndx member is SHN_UNDEF and the symbol is of type STT_FUNC and the st_value member is not zero, the dynamic linker recognizes the entry as special and uses the st_value member as the symbol's address.

Otherwise, the dynamic linker considers the symbol to be undefined within the executable file and continues processing.

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

4.2.4 Procedure Linkage Table

Procedure linkage tables (PLTs) are used to redirect function calls between the executables and shared objects or between shared objects.

The PLT is designed to permit lazy or deferred symbol resolution at run time, and to allow for dynamic run-time patching and upgrading of library code.

Several PLT entries may exist for the same function, corresponding to the calls and references from several different libraries, but for each program there is exactly one dominant PLT entry per function, which serves as the formal function address for the function. All pointer comparisons use this PLT address when referencing the function. The dominant PLT entry is the first PLT entry processed by the dynamic linker. The dynamic linker resolves all subsequent references to the function to this first address.

The PLT may be subsumed within the .got section, or divided into parts: A read-only executable part (.plt) The .plt portion of a PLT in such an arrangement consists of an array of 12-byte entries, one entry for each function requiring PLT linkage. A read-write data part (.pltgot) The .pltgot portion is a subsection of the GOT table and contains one four-byte address per PLT entry. The purpose of a .pltgot is to isolate the PLT-specific .got entries from the rest of the .got; in this arrangement, no part of the .got is ever marked read only. If a PLT entry is required by the operating system, a static linker may generate a fixed sequence of code in the read-only part of the PLT that loads the address and jumps to it.

The following example lists a permissible assembly-language definition of a PLT entry.

Example: PLT Entry in ARCv2 Assembly Language

```
ld %r12,[%pcl,func@gotpc]
j [%r12]
mov %r12,%pcl
```

This PLT entry sequence occupies 16 bytes of storage. In the preceding example, `func@gotpc` represents the PC-relative offset of the location in the GOT (or PLTGOT) that contains the actual address of the function or the address of the code stub that transfers control to the dynamic linker.

When executed, the PLT code loads the actual address of the function into r12 from the GOT. It then jumps through r12 to its destination. As it jumps, the delay-slot instruction loads r12 with the current value of the 32-bit-aligned PC address for identification. The PLT-entry PC address identifies the function called by allowing the lazy loader to calculate the index into the PLT, which also corresponds to the index of the relocation in the .rela.plt relocation section. The writable GOT or PLTGOT entry is initialized by the dynamic linker when the object is first loaded into memory. At first it is initialized to the special code stub that saves the volatile registers and calls the dynamic linker. The first time the function is called, the dynamic linker loads, links, and resolves the GOT or PLTGOT entry to point to the actual loaded function for subsequent calls.

The first entry in the PLT is reserved and is used as a reference to transfer control to the dynamic linker. At program load time, each GOT or PLTGOT entry is set to PLT[0], which is a hard-coded jump to the dynamic-link stub routine.

The code residing at the beginning of the PLT occupies 24 bytes of storage. The code is the equivalent of the following:

```
ld r11, [pcl, (GOT+4)@gotpc] ; module info stored by dynamic loader
ld r10, [pcl, (GOT+8)@gotpc] ; dynamic loader entry point
j [r10]
```

A relocation table (.rela.plt) is associated with the PLT. The DT_JMPREL entry in the dynamic section gives the location of the first relocation entry. The relocation table's entries parallel the PLT entries in a one-to-one correspondence. That is, relocation table entry 1 applies to PLT entry 1, and so on. The relocation type for each entry is R_ARC_JMP_SLOT. The relocation offset shall specify the address of the GOT or PLTGOT entry associated with the function, and the symbol table index shall reference the function's symbol in the .dynsym symbol table. The dynamic linker locates the symbol referenced by the R_ARC_JMP_SLOT relocation. The value of the symbol is the address of the first instruction of the function's PLT entry.

The dynamic linker can resolve the procedure linkage table relocations lazily, resolving them only when they are needed. Doing so might reduce program startup time. 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_ARC_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, some 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 such unpredictability.

If an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker terminates the program. Under lazy binding, this might occur at arbitrary times. Some applications cannot tolerate such unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.