64-bit ELF Object File Specification

Version 2.4

Jim Dehnert

MIPS Technologies /

Silicon Graphics Computer Systems

Caveat: This document represents work in progress. It is incomplete and subject to change. In particular, lists of constants, sections, and attributes are may be incomplete or inaccurate in detail. Reference to the header files ***elf.h*** and ***sys/elf.h*** is recommended before reliance on the information herein.

Section 1 Introduction

This document specifies the format of MIPS object files for 64-bit code. We assume as a basis the documents [ABI32], [ABI32M], and [ABI64]. In addition, Silicon Graphics uses the DWARF debugging information format as specified in [DWARF]. Information in those documents should be considered valid unless contradicted here.

The remainder of this section summarizes our objectives, approach, and open issues. Section 2 corresponds to ABI Section 4, describing ELF-64 object files, and Section 3 corresponds to ABI Section 5, describing program loading and dynamic linking. Section 4 describes the 64-bit archive file format (from MIPS ABI Section 7), though it is not technically part of ELF.

1.1 Objectives

The objectives of this format definition fall in two categories. The first is simply to extend the base 32-bit ELF format, by increasing field sizes where appropriate, so that there will be no problems dealing with very large 64-bit programs. The second is to extend the kinds of information included in the object file to facilitate new features.

1. Remove 32-bit constraints on object file sizes.
2. Support checking and potential conversion of subprogram call interfaces.
3. Support efficient and reliable application of performance monitoring tools like pixie and object code optimization tools like cord.
4. Support efficient and reliable debugging facilities.

1.2 Approach

We start from the basis of the System V ABI and the MIPS symbol table format, extending them in the obvious ways to support very large objects. We will then add additional information (generally new sections) to support the extended objectives described above. We generally recognize three levels of support:

1. Some information is required for all ABI-compliant object files.
2. Some information is optional, but its absence will prevent use of system functionality, e.g. cache optimization reordering, etc.
3. Some information is entirely optional; its absence interferes only with functionality unrelated to program construction, e.g. debugging, performance measurement, etc.

To clarify the distinction between these levels, we have defined a new section header flag, **SHF_MIPS_NOSTRIP**, which is applied to sections in the first two levels. The intent is that a ***strip(1)*** tool or its equivalent should never remove these sections by default, and should warn the user if their removal is explicitly requested.

We have followed a number of principles and assumptions in this specification:

1. Sections expected to be used by runtime facilities, e.g. stack traceback, have the **SHF_ALLOC** attribute. Such sections also have symbols automatically generated by the linker which may be used to reference them in code (see Section 3.1.2). This allows simple virtual addressing in the process address space for any required access.
2. Sections are NOT given the **SHF_WRITE** attribute simply because rld may need to relocate their contents. We assume that rld can request write access and change it back if necessary, and prefer this approach for a more robust runtime environment.
3. We require that the executable code for a single executable or DSO will never be larger than 256MB, and that it will never be loaded across a 256MB boundary. This requirement allows various beneficial assumptions about valid addressing code within an executable/DSO, and also allows the use of single-word addresses (to be interpreted relative to the containing 4GB address range) for code in an object file.

Finally, this is intended to be a permissive specification. Usage of the features described should generally be viewed as permitted in any combination with a reasonably unambiguous interpretation unless it is forbidden. It is likely, of course, that new usage will uncover (and break) implicit assumptions in tools from time to time, and adding further restrictions may be the most appropriate solution, but the bias should be towards fixing the tools to allow reasonable practices.

1.3 Conventions

In all of the tables in this document, unshaded information is that derived directly from external documents (i.e. the generic ABI and the DWARF specification), lightly shaded information is derived directly from 32-bit MIPS specifications (ELF, header files), and heavily shaded information is new or substantially changed from these existing formats.

We use the usual wording to describe requirements, distinguishing between *must* (mandatory), *should* (recommended), and *may* (allowed).

1.4 Open Issues

ISSUE !

Unresolved issues and missing information are marked in this document by the symbol at the left. The most significant ones at this time are:

1. Should there be a distinct EK_PBEGIN event in case a program unit does not begin with an entrypoint (Section 2.10)?
2. Should memory events be segregated to a distinct events section so that tools which use the default events and those which use the memory events aren't impacted by the other data (Section 2.10)?

1.5 Changes from Version 2.0

The following have been changed from version 2.0 of this specification:

1. Added a description of LEB128/ULEB128 formats.
2. Added **EF_MIPS_OPTIONS_FIRST** and **EF_MIPS_ARCH_ASE** ELF flags.
3. Added Table 15 describing special reserved values of the *st_shndx* field.
4. Added definitions of **OHW_R8KPFETCH** and **OHW_R5KEOP** masks in the Hardware Patch Option Descriptor. Added new Hardware AND/OR Patch Option Descriptors.
5. Added new **ODK_GP_GROUP** and **ODK_IDENT** option descriptors.
6. Note that **R_MIPS_PJUMP** requires that ***ld*** check that the symbol is not preemptible before performing the relocation.
7. Added definitions of event kinds **EK_LTR_FCALL** and **EK_PCREL_GOT0**. Fixed values of kinds **EK_MEM_***.
8. Fixed definition of content kind **CK_GP_GROUP**.
9. Added section on Shared Object Dependencies describing default search paths for DSOs and environment variables to override them.
10. Added sections defining the **.msym** and **.conflict** sections for the quickstart discusion (Section 3.8).
11. Added **DT_MIPS_INTERFACE_SIZE** and **DT_MIPS_RLD_TEXT_RESOLVE_ADDR** descriptions.
12. Added missing **DT_MIPS_FLAGS** flags.
13. Added documentation of the **.liblist** section (Section 3.8.2).
14. Added documentation of the **.MIPS.symlib** section (Section 3.8.4).
15. Corrected default library search paths in Section 3.4.

16. Added **SHN_MIPS_LCOMMON** (thread-local common) and **SHN_MIPS_LCOMMON** (thread-local undefined) symbol documentation in Section 2.5.
17. Added **STO_OPTIONAL** symbol documentation in Section 2.5.
18. Added split common symbol documentation in Section 2.5 (and Section 2.5.1).
19. Fixed **EK_PCREL_GOT0** definition in Section 2.10.
20. Clarified **CK_GP_GROUP** description in Section 2.12.
21. Add comment to **ODK_PAD** option descriptor, allowing implementations to require that ***ld*** do all padding rather than leaving some to ***rld*** (Section 2.8).
22. Added **ODK_PAGESIZE** proposal (Section 2.8.10).

Section 2 ELF-64 Object File Format

The ELF-64 object file format is based on the document [ABI64]. The relevant declarations are contained in the header file ***/usr/include/elf.h***. (Note that ***/usr/include/sys/elf.h*** is logically part of ***/usr/include/elf.h***, and is the actual location of most of these declarations.)

ELF-32

We call attention to the [ABI32] requirement that all data structures be naturally aligned in the file (see p. 4-3). This implies that all headers, sections, and other major components must be 8-byte aligned, with padding if necessary to accomplish this. Although 4-byte alignment is probably adequate currently for ELF-32, 8-byte alignment should also be used there to avoid future extension problems. (Note, however, that the DWARF specification fundamentally assumes a byte-stream format. Therefore, the data structures contained in some DWARF sections will violate the alignment requirement. In addition, the .MIPS.events and .MIPS.content sections have packed byte-stream contents for compactness.)

When defining structure fields which are smaller than a convenient storage unit (e.g. single-bit flags), we have used C's bitfield notation (e.g. Elf64_Word:1). The intent is to specify layout equivalent to big-endian bitfields, but the actual structure declarations in header files should use masks and shifts to access them. This avoids problems with byte swapping between big- and little-endian hosts.

2.1 Infrastructure

ELF-64 is defined in terms of the types in Table 1:

Table 1 ELF-64 Data Types

Name	Size	Alignment	Purpose
Elf64_Addr	8	8	Unsigned program address
Elf64_Half	2	2	Unsigned small integer
Elf64_Off	8	8	Unsigned file offset
Elf64_Sword	4	4	Signed medium integer
Elf64_Sxword	8	8	Signed large integer

Table 1 ELF-64 Data Types

Name	Size	Alignment	Purpose
Elf64_Word	4	4	Unsigned medium integer
Elf64_Xword	8	8	Unsigned large integer
Elf64_Byte	1	1	Unsigned tiny integer
Elf64_Section	2	2	Section index (unsigned)

There are places in ELF-64 files where fundamental data types must be encoded, for instance in subprogram interface descriptors. We generally use the constants in Table 2 to identify them, based on DWARF version 1 (but not identical).

Table 2 Fundamental Data Types

Name(s)	Value	Comments
FT_unknown	`0x0001`	unknown type
FT_signed_char	`0x0001`	8-bit signed character
FT_unsigned_char	`0x0002`	8-bit unsigned character
FT_signed_short	`0x0003`	16-bit signed short integer
FT_unsigned_short	`0x0004`	16-bit unsigned short integer
FT_signed_int32	`0x0005`	32-bit signed integer
FT_unsigned_int32	`0x0006`	32-bit unsigned integer
FT_signed_int64	`0x0007`	64-bit signed integer
FT_unsigned_int64	`0x0008`	64-bit unsigned integer
FT_pointer32	`0x0009`	32-bit pointer
FT_pointer64	`0x000a`	64-bit pointer
FT_float32	`0x000b`	32-bit floating point (IEEE)
FT_float64	`0x000c`	64-bit floating point (IEEE)
FT_float128	`0x000d`	128-bit floating point
FT_complex64	`0x000e`	64-bit complex floating point
FT_complex128	`0x000f`	128-bit complex floating point

Table 2 Fundamental Data Types

Name(s)	Value	Comments
FT_complex256	`0x0010`	256-bit complex floating point
FT_void	`0x0011`	void
FT_bool32	`0x0012`	32-bit Boolean (TRUE or FALSE)
FT_bool64	`0x0013`	64-bit Boolean (TRUE or FALSE)
FT_label32	`0x0014`	32-bit label (address)
FT_label64	`0x0015`	64-bit label (address)
FT_struct	`0x0020`	structure (record)
FT_union	`0x0021`	union (variant)
FT_enum	`0x0022`	enumerated type
FT_typedef	`0x0023`	typedef
FT_set	`0x0024`	Pascal: set
FT_range	`0x0025`	Pascal: subrange of integer
FT_member_ptr	`0x0026`	C++: member pointer
FT_virtual_ptr	`0x0027`	C++: virtual pointer
FT_class	`0x0028`	C++: class

The fundamental types in Table 2 may be modified by the qualifiers in Table 3 below, also based on [DWARF-1]:

Table 3 Type Qualifiers

Name	Value	Comments
MOD_pointer_to	`0x01`	pointer to base type
MOD_reference_to	`0x02`	C++: reference to base type
MOD_const	`0x03`	const
MOD_volatile	`0x04`	volatile
MOD_lo_user	`0x80`	first MIPS-specific modifier
MOD_function	`0x80`	function returning base type
MOD_array_of	`0x81`	array of base type

Table 3 Type Qualifiers

Name	Value	Comments
MOD_hi_user	`0xff`	last MIPS-specific modifier

The data structures in the **.MIPS.events** and **.MIPS.content** sections use compressed types from the [DWARF] specification, named LEB128 and ULEB128, for *(Unsigned) Little-Endian Base 128* numbers. These are "little endian" only in the sense that they avoid using space to represent the "big" end of an integer when the big end is all zeroes (unsigned) or sign extension bits (signed).

ULEB128 numbers are encoded as follows: start at the low-order end of an unsigned integer and chop it into 7-bit chunks. Place each chunk into the low-order 7 bits of a byte. Typically, several of the high-order bytes will be zero (unsigned) or copies of the sign bit (signed) — discard them. Emit the remaining bytes in a stream, starting with the low-order byte; set the high order bit on each byte except the last emitted byte. The high bit of zero on the last byte indicates to the decoder that it has encountered the last byte.

2.2 ELF-64 Header

The header format is as defined in [ABI64]; it is reproduced here for reference purposes.

Table 4 ELF-64 Header Structure

Field Name	Type	Comments
e_ident[EI_NIDENT]	**unsigned char**	See Table 5
e_type	**Elf64_Half**	See [ABI32]
e_machine	**Elf64_Half**	Machine (**EM_MIPS** = 8)
e_version	**Elf64_Word**	File format version
e_entry	**Elf64_Addr**	Process entry address
e_phoff	**Elf64_Off**	Program header table file offset
e_shoff	**Elf64_Off**	Section header table file offset

Table 4 ELF-64 Header Structure

Field Name	Type	Comments
e_flags	**Elf64_Word**	Flags — see Table 6
e_ehsize	**Elf64_Half**	ELF header size (bytes)
e_phentsize	**Elf64_Half**	Program header entry size
e_phnum	**Elf64_Half**	Number of program headers
e_shentsize	**Elf64_Half**	Section header entry size
e_shnum	**Elf64_Half**	Number of section headers
e_shstrndx	**Elf64_Half**	Section name string table section header index

The structure of the *e_ident* field is given by Table 5.

Table 5 ELF-64 Header: *e_ident*[ ] Contents

Offset Name	Index	Value or Interpretation
EI_MAG0-3	`0-3`	Magic string: 0x7f, 'E', 'L', 'F'
EI_CLASS	`4`	Class of format: ELFCLASS64 = 2
EI_DATA	`5`	Endianness: ELFDATAMSB = 2
EI_VERSION	`6`	Version of format: EV_CURRENT = 1
EI_PAD	`7-15`	Reserved, must be zero

Flags currently defined for the *e_flags* field are given by .

Table 6 ELF-64 Header: Processor-Specific Flags in *e_flags*

Flag Name	Value	Comments
EF_MIPS_NOREORDER	`0x00000001`	At least one .noreorder assembly directive appeared in a source contributing to the object
EF_MIPS_PIC	`0x00000002`	This file contains position-independent code

Table 6 ELF-64 Header: Processor-Specific Flags in *e_flags*

Flag Name	Value	Comments
EF_MIPS_CPIC	`0x00000004`	This file's code follows standard conventions for calling position-independent code
EF_MIPS_UCODE	`0x00000010`	This file contains UCODE (obsolete)
EF_MIPS_ABI2	`0x00000020`	This file follows the MIPS III 32-bit ABI. (Its **EI_CLASS** will be **ELFCLASS32**.)
EF_MIPS_OPTIONS_FIRST	`0x00000080`	This **.MIPS.options** section in this file contains one or more descriptors, currently types **ODK_GP_GROUP** and/or **ODK_IDENT**, which should be processed first by ***ld***.
EF_MIPS_ARCH_ASE	`0x0f000000`	Application-specific architectural extensions used by this object file:
EF_MIPS_ARCH_ASE_MDMX	`0x08000000`	Uses MDMX multimedia extensions
EF_MIPS_ARCH_ASE_M16	`0x04000000`	Uses MIPS-16 ISA extensions
EF_MIPS_ARCH	`0xf0000000`	Architecture assumed by code in this file, given by the value of the 4-bit field selected by the mask: MIPS I (0), MIPS II (1), MIPS III (2), MIPS IV (3)

NOTE: PIC code is inherently CPIC, and may or may not set **EF_MIPS_CPIC**.

2.3 ELF-64 Section Header

The section header format is as defined in [ABI64]; it is reproduced here for reference purposes.

Table 7 Section Header Structure *(Elf64_Shdr)*

Name	Type	Description
sh_name	**Elf64_Word**	Section name (index into section header string table section)
sh_type	**Elf64_Word**	Section type: see Table 8
sh_flags	**Elf64_Xword**	Section flags: see Table 9
sh_addr	**Elf64_Addr**	Address of first byte, or zero
sh_offset	**Elf64_Off**	File offset of section
sh_size	**Elf64_Xword**	Section's size in bytes
sh_link	**Elf64_Word**	Table index link: section-specific
sh_info	**Elf64_Word**	Extra information: section-specific
sh_addralign	**Elf64_Xword**	Address alignment constraint
sh_entsize	**Elf64_Xword**	Size of fixed-size entries in section, or zero

The valid section types for section header field *sh_type* are given in Table 7 below. Shaded types are MIPS-specific; heavily shaded types are new in this specification.

Table 8 Section Types

Name	Value	Description
SHT_NULL	0	Inactive section.
SHT_PROGBITS	1	Information defined by the program
SHT_SYMTAB	2	Symbol table (one per object file)
SHT_STRTAB	3	String table (multiple sections OK)
SHT_RELA	4	Relocation with explicit addends

Table 8 Section Types

Name	Value	Description
SHT_HASH	5	Symbol hash table (one per object)
SHT_DYNAMIC	6	Dynamic linking information
SHT_NOTE	7	Vendor-specific file information
SHT_NOBITS	8	Section contains no bits in object file
SHT_REL	9	Relocation without explicit addends
SHT_SHLIB	10	Reserved — non-conforming
SHT_DYNSYM	11	Dynamic linking symbol table (one)
SHT_LOPROC	0x70000000	First processor-specific type
SHT_HIPROC	0x7fffffff	Last processor-specific type
SHT_LOUSER	0x80000000	First application-specific type
SHT_HIUSER	0x8fffffff	Last application-specific type
SHT_MIPS_LIBLIST	0x70000000	DSO library information used in link
SHT_MIPS_MSYM	0x70000001	MIPS symbol table extension
SHT_MIPS_CONFLICT	0x70000002	Symbols conflicting with DSO-defined symbols
SHT_MIPS_GPTAB	0x70000003	Global pointer table
SHT_MIPS_UCODE	0x70000004	Reserved
SHT_MIPS_DEBUG	0x70000005	Reserved (obsolete debug information)
SHT_MIPS_REGINFO	0x70000006	Register usage information
SHT_MIPS_PACKAGE	0x70000007	OSF reserved
SHT_MIPS_PACKSYM	0x70000008	OSF reserved
SHT_MIPS_RELD	0x70000009	Dynamic relocation?
unused	0x7000000a	
SHT_MIPS_IFACE	0x7000000b	Subprogram interface information
SHT_MIPS_CONTENT	0x7000000c	Section content classification
SHT_MIPS_OPTIONS	0x7000000d	General options
SHT_MIPS_DELTASYM	0x7000001b	Delta C++: symbol table
SHT_MIPS_DELTAINST	0x7000001c	Delta C++: instance table
SHT_MIPS_DELTACLASS	0x7000001d	Delta C++: class table
SHT_MIPS_DWARF	0x7000001e	DWARF debug information

Table 8 Section Types

Name	Value	Description
SHT_MIPS_DELTADECL	`0x7000001f`	Delta C++: declarations
SHT_MIPS_SYMBOL_LIB	`0x70000020`	Symbol-to-library mapping.
SHT_MIPS_EVENTS	`0x70000021`	Event locations
SHT_MIPS_TRANSLATE	`0x70000022`	???
SHT_MIPS_PIXIE	`0x70000023`	Special pixie sections
SHT_MIPS_XLATE	`0x70000024`	Address translation table[a]
SHT_MIPS_XLATE_DEBUG	`0x70000025`	SGI internal address translation table[a]
SHT_MIPS_WHIRL	`0x70000026`	Intermediate code
SHT_MIPS_EH_REGION	`0x70000027`	C++ exception handling region info
SHT_MIPS_XLATE_OLD	`0x70000028`	Obsolete
SHT_MIPS_PDR_EXCEPTION	`0x70000029`	Runtime procedure descriptor table exception information (ucode)

[a] **SHT_MIPS_XLATE** contains translation data table as created by the xlate library from within cord/pixie, for use by debuggers and other tools which need to know how to map addresses in the binary text to addresses in the debug information. See the system header file ***/usr/lib/include/Xlate.h***. **SHT_MIPS_XLATE_DEBUG** has the same data format as **SHT_MIPS_XLATE**.

The section attribute flags defined for section header field *sh_flags* are given in the table below. Again, light shading indicates MIPS-specific flags, and heavier shading new flags.

Table 9 Section Attribute Flags

Name	Value	Description
SHF_WRITE	`0x1`	Section writable during execution
SHF_ALLOC	`0x2`	Section occupies memory
SHF_EXECINSTR	`0x4`	Section contains executable instructions
SHF_MASKPROC	`0xf0000000`	Reserved for processor-specific flags
SHF_MIPS_GPREL	`0x10000000`	Section must be part of global data area.[c]

Table 9 Section Attribute Flags

Name	Value	Description
SHF_MIPS_MERGE [a]	0x20000000	Section data should be merged to eliminate duplication
SHF_MIPS_ADDR	0x40000000	Section data is addresses by default (see Section 2.12). Address size to be inferred from section entry size.
SHF_MIPS_STRING	0x80000000	Section data is string data by default (see Section 2.12).
SHF_MIPS_NOSTRIP	0x08000000	Section data may not be stripped
SHF_MIPS_LOCAL	0x04000000	Section data local to process [b,c]
SHF_MIPS_NAMES	0x02000000	Linker must generate implicit hidden weak names — see Section 3.1.2
SHF_MIPS_NODUPE	0x01000000	Section contains text/data which may be replicated in other sections. Linker must retain only one copy.

[a] For a merged section, the **SH_INFO** value is the size (in bytes) of the objects to be merged. Such sections should not be writable.

[b] Local (**SHF_MIPS_LOCAL**) sections are for multi-process programs sharing an address space. They must be copied for each process which attempts to write to them. This copying does not occur until after the second process is spawned, so that the initial process can perform dynamic initialization common to all processes' copies of the section. This attribute replaces the predefined COFF **lcldta** and **.lbss** sections.

[c] **SHF_MIPS_LOCAL** and **SHF_MIPS_GPREL** are mutually exclusive, i.e. a local data section may not be placed in the short gp-relative data area.

A number of special sections are predefined, with standard names and attributes. Note, however, that an ELF producer may create arbitrary sections with arbitrary names and attributes, and may generate the predefined sections with additional attributes. For example, executable code may be generated in multiple sections with arbitrary names, not just in .text.

Table 10 Special Sections

Name	Type	Default Attributes
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE

Table 10 Special Sections

Name	Type	Default Attributes
.comment	`SHT_PROGBITS`	none (by default)
.data	`SHT_PROGBITS`	SHF_ALLOC + SHF_WRITE
.data1	`SHT_PROGBITS`	(gABI, not used by MIPS)
.debug	`SHT_PROGBITS`	(gABI, not used by MIPS)
.dynamic	`SHT_DYNAMIC`	SHF_ALLOC [a]
.dynstr	`SHT_STRTAB`	SHF_ALLOC
.dynsym	`SHT_DYNSYM`	SHF_ALLOC
.fini	`SHT_PROGBITS`	SHF_ALLOC + SHF_EXECINSTR
.got	`SHT_PROGBITS`	SHF_ALLOC + SHF_WRITE + SHF_MIPS_GPREL
.hash	`SHT_HASH`	SHF_ALLOC
.init	`SHT_PROGBITS`	SHF_ALLOC + SHF_EXECINSTR
.interp	`SHT_PROGBITS`	SHF_ALLOC
.line	`SHT_PROGBITS`	SHF_ALLOC
.note	`SHT_NOTE`	none (by default)
.plt	`SHT_PROGBITS`	(gABI, not used by MIPS)
.rel*name*	`SHT_REL`	none (by default), see [ABI32]
.rela*name*	`SHT_RELA`	none (by default), see [ABI32]
.rodata	`SHT_PROGBITS`	SHF_ALLOC
.rodata1	`SHT_PROGBITS`	(gABI, not used by MIPS)
.shstrtab	`SHT_STRTAB`	(gABI, not used by MIPS)
.strtab	`SHT_STRTAB`	none
.symtab	`SHT_SYMTAB`	none
.text	`SHT_PROGBITS`	SHF_ALLOC + SHF_EXECINSTR
.MIPS.Addrs	`SHT_MIPS_PIXIE`	none
.MIPS.Binmap	`SHT_MIPS_PIXIE`	none
.MIPS.compact_rel	`SHT_MIPS_COMPACT`	none
.conflict [c]	`SHT_MIPS_CONFLICT`	SHF_ALLOC
.MIPS.content*name*	`SHT_MIPS_CONTENT`	SHF_ALLOC+ SHF_MIPS_NOSTRIP
.debug_abbrev	`SHT_MIPS_DWARF`	none (generic DWARF section)

Table 10 Special Sections

Name	Type	Default Attributes
.debug_aranges	SHT_MIPS_DWARF	none (generic DWARF section)
.debug_frame	SHT_MIPS_DWARF	SHF_MIPS_NOSTRIP (generic DWARF section)
.debug_funcnames	SHT_MIPS_DWARF	none (generic DWARF section)
.debug_info	SHT_MIPS_DWARF	none (generic DWARF section)
.debug_line	SHT_MIPS_DWARF	none (generic DWARF section)
.debug_loc	SHT_MIPS_DWARF	none (generic DWARF section)
.debug_pubnames	SHT_MIPS_DWARF	none (generic DWARF section)
.debug_str	SHT_MIPS_DWARF	none (generic DWARF section)
.debug_typenames	SHT_MIPS_DWARF	none (MIPS DWARF section)
.debug_varnames	SHT_MIPS_DWARF	none (MIPS DWARF section)
.debug_weaknames	SHT_MIPS_DWARF	none (MIPS DWARF section)
.dynamic	SHT_DYNAMIC	SHF_ALLOC
.MIPS.events*name*	SHT_MIPS_EVENTS	SHF_ALLOC+SHF_MIPS_NOSTRIP
.gptab*name* [b]	SHT_MIPS_GPTAB	none
.MIPS.Graph	SHT_MIPS_PIXIE	none
.MIPS.interfaces	SHT_MIPS_IFACE	SHF_ALLOC+SHF_MIPS_NOSTRIP
.MIPS.lbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_MIPS_LOCAL [d]
.MIPS.ldata	SHT_PROGBITS	
.lib	obsolete	
.liblist [c]	SHT_MIPS_LIBLIST	SHF_ALLOC
.lit4 [b]	SHT_PROGBITS	SHF_ALLOC + SHF_MIPS_MERGE + SHF_MIPS_GPREL [a]
.lit8 [b]	SHT_PROGBITS	SHF_ALLOC + SHF_MIPS_MERGE + SHF_MIPS_GPREL [a]
.MIPS.lit16 [b]	SHT_PROGBITS	SHF_ALLOC + SHF_MIPS_MERGE + SHF_MIPS_GPREL [a]
.MIPS.Log	SHT_MIPS_PIXIE	none
.MIPS.Map	SHT_MIPS_PIXIE	none
.mdebug	obsolete	to be replaced by DWARF sections
.MIPS.options	SHT_MIPS_OPTIONS	SHF_ALLOC+SHF_MIPS_NOSTRIP

Table 10 Special Sections

Name	Type	Default Attributes
.msym	`SHT_MIPS_MSYM`	SHF_ALLOC
.MIPS.Graph	`SHT_MIPS_PIXIE`	none
.MIPS.Perf_argtrace	`SHT_MIPS_PIXIE`	none
.MIPS.Perf_bb_offsets	`SHT_MIPS_PIXIE`	none
.MIPS.Perf_call_graph	`SHT_MIPS_PIXIE`	none
.MIPS.Perf_function	`SHT_MIPS_PIXIE`	none
.MIPS.Perf_table	`SHT_MIPS_PIXIE`	none
.MIPS.Perf_weak_names	`SHT_MIPS_PIXIE`	none
.rel.dyn	`SHT_REL`	SHF_ALLOC
.reld*name*	`SHT_RELA`	SHF_ALLOC
.sbss [b]	`SHT_NOBITS`	SHF_ALLOC + SHF_MIPS_GPREL + SHF_WRITE
.sdata [b]	`SHT_PROGBITS`	SHF_ALLOC + SHF_MIPS_GPREL + SHF_WRITE
.srdata [b]	`SHT_PROGBITS`	SHF_ALLOC + SHF_MIPS_GPREL
.MIPS.symlib	`SHT_MIPS_SYMBOL_LIB`	SHF_ALLOC
.MIPS.translate	`SHT_PROGBITS`	SHF_MIPS_NOSTRIP + SHF_ALLOC (non-shared only)
.ucode	`SHT_MIPS_UCODE`	none (obsolete)

[a] [ABI32M] specifies these sections with SHF_WRITE as well. Why?
[b] A MIPS ABI-64 compiliant system must support these sections.
[c] A MIPS ABI-64 compliant system must recognize, but may choose to ignore, these sections. However, if either is supported, both must be.
[d] The **.lbss** and **lcldata** sections can be replaced with arbitrary sections having the **SHF_MIPS_LOCAL** attribute; they will no longer be recognized by the system strictly based on name.

Linker Processing

We expect more use of non-predefined sections in the future to achieve greater control over process memory allocation. The linker is expected to combine sections with matching name and attributes, then groupings with matching attributes, and finally groupings with consistent attributes (e.g. all **SHT_MIPS_GPREL** sections, or all read-only sections). The precise rules will be defined in Section 3.

2.4 String Table

This section type is unchanged from [ABI32]. A String Table section has the following attributes:

name	**.strtab**
sh_type	**SHT_STRTAB**
sh_link	**SHN_UNDEF**
sh_info	0
sh_flags	**SHF_ALLOC**
requirements	see **.symtab**

2.5 Symbol Table

A symbol table section is unchanged from [ABI32] except for the types of some of its fields.

A symbol table section and its associated string table section must be present for any executable file with DSO dependencies, or for any DSO. It is permissible to remove from it symbols resolved within itself if they are not preemptible (protected) and not visible outside this object. (All defined symbols in an executable file are protected, but symbols must have been explicitly declared protected in a DSO, and hidden in either a DSO or an executable. See the discussion associated with Table 14 for definitions of these terms.)

A Symbol Table section has the following attributes:

name	**.symtab**
sh_type	**SHT_SYMTAB**
sh_link	Section header index of the associated string table
sh_info	0
sh_flags	**SHF_ALLOC**
requirements	see discussion above

The structure of a symbol table item is given by Table 11 below.

Table 11 ELF-64 Symbol Table Structure

Name	Type	Comments
st_name	**Elf64_Word**	Name's index into string table
st_info	**Elf64_Byte**	Symbol type and binding: see Table 12 and Table 13
st_other	**Elf64_Byte**	Other info: see Table 14
st_shndx	**Elf64_Section**	Index of section where defined
st_value	**Elf64_Addr**	Symbol value
st_size	**Elf64_Xword**	Symbol size

The high-order nibble of the st_info field specifies the symbol's binding (see Table 12), and the low-order nibble specifies its type (see Table 13).

Table 12 Symbol Binding (*ELF32_ST_BIND*)

Constant Name	Value	Comments
STB_LOCAL	0	Not visible outside object file where defined
STB_GLOBAL	1	Visible to all object files. Multiple definitions cause errors. Force extraction of defining object from archive file.
STB_WEAK	2	Visible to all object files. Ignored if STB_GLOBAL with same name found. Do not force extraction of defining object from archive file. Value is 0 if undefined.
STB_LOPROC	13	First processor-specific binding
STB_SPLIT_COMMON	13	Split common symbol. See Section 2.5.1.
STB_HIPROC	15	Last processor-specific binding

Table 13 Symbol Type (*ELF32_ST_TYPE*)

Constant Name	Value	Comments
STT_NOTYPE	0	Not specified
STT_OBJECT	1	Data object: variable, array, etc.
STT_FUNC	2	Function or other executable code
STT_SECTION	3	Section. Exists primarily for relocation
STT_FILE	4	Name (pathname?) of the source file associated with object. Binding is STT_LOCAL, section index is SHN_ABS, and it precedes other STB_LOCAL symbols if present
STT_LOPROC	13	First processor-specific type
STT_HIPROC	15	Last processor-specific type

The binding type of a symbol is used to control its visibility, as well as resolution in the case of multiple definitions, between the relocatable objects comprising a program. This semantics is extended to interfaces between an executable and/or DSOs unchanged — it is simply interpreted by the dynamic linker instead of the static linker. In order to allow independent control of interfaces between executable and DSOs without affecting the binding type semantics within them, information in the *st_other* field, as given in Table 14 below, is used to specify visibility and accessibility of symbols outside the containing executable/DSO, which we term the symbol *export class*.

- By default, global, weak, or common symbols are ***preemptible***, i.e. they may be preempted by definitions of the same name elsewhere.
- Symbols defined in the current component are ***protected*** if they are visible outside but not preemptible, meaning that any reference to such a symbol from within the defining executable or DSO must resolve to the local definition even if there are definitions in other executables or DSOs which would normally preempt it.
- Symbols defined in the current component are ***hidden*** if their names are not visible outside — such symbols are necessarily protected, and this attribute may be used to control the external interface of a DSO, but such objects may still be referenced from outside if an address is passed outside as a pointer.

- Symbols are ***internal*** if their addresses are not passed outside, e.g. static C functions whose address is never taken.

The visibility semantics of these attributes also allow various optimizations. Whereas care must be taken to maintain position-independence and proper GOT usage for references to and definitions of symbols which might be preempted by or referenced from other DSOs, these restrictions all allow references from the same executable/DSO to make stricter assumptions about the definitions. References to protected symbols (and hence to hidden or internal symbols) may be optimized by using absolute addresses in executables or by assuming addresses to be relatively nearby. Internal functions do not normally require **gp** establishment code because they will always be entered from the same executable/DSO with the correct **gp** already set up**.**

Linker Processing

None of these attributes affects resolution of symbols within an executable or DSO during static linking — such resolution is controlled by the binding type. (However, if the static linker references symbol definitions in other DSOs during link time, it is constrained by their export classes.) Once the static linker has chosen its resolution, these attributes impose two requirements, both based on the fact that various references in the code being linked may have been optimized to take advantage of the attributes. First, all of these attributes imply that a symbol must be defined in the current DSO/executable. If a symbol with one of these attributes has no definition within the executable/DSO being linked, then it must be resolved to allocated space if common, resolved to zero if weak, or an error reported otherwise. Second, if any reference to, or definition of, a name is a symbol with one of these attributes, the attribute must be propagated to the resolving symbol in the linked object.

Table 14 *st_other* Field Masks

Constant Name	Value	Comments
STO_EXPORT	3	DSO export class — one of:
STO_DEFAULT	0	Default: **STB_GLOBAL** or **STB_WEAK** are preemptible, **STB_LOCAL** are hidden.
STO_INTERNAL	1	Not referenced outside executable/DSO
STO_HIDDEN	2	Not visible outside executable/DSO
STO_PROTECTED	3	Not preemptible

Table 14 *st_other* Field Masks

Constant Name	Value	Comments
STO_OPTIONAL	4	Symbol is optional. If no definition is available at runtime, it is resolved to the symbol **_RLD_MISSING**.

Normally in relocatable files, a symbol's value refers to its offset within the section specified by the *st_shndx* field. Its value will therefore be adjusted as the section moves during relocation. Certain special section index values imply other semantics, as described in Table 15:

Linker Processing

Resolution rules for optional symbols (**STO_OPTIONAL**) are as follows. The static linker (***ld***) should convert a reference to an optional definition (i.e. in another DSO) to an optional reference. The merge of an optional and a non-optional reference becomes optional. The optional type is ignored for **SHN_COMMON** and **SHN_ACOMMON** symbols, and does not affect stub generation and resolution. In the runtime linker (***rld***), unresolved optional symbols are silently resolved to the reserved symbol **_RLD_MISSING**, defined in ***libc.so.1***. Optional references do not trigger the loading of delay-loaded libraries. Therefore, optional references may go unresolved until some other event triggers the loading of the delay-loaded library, and one may not check the availability of an optional feature found in a delay-loaded library until some other event has forced the library to be loaded.

Table 15 *st_shndx* Field Special Values

Name	Value	Semantics
SHN_UNDEF	0	The symbol is undefined. Its value will be determined by its appearance as a defined symbol in another object file.
SHN_LORESERVE	0xff00	Section indices between **SHN_LORESERVE** and **SHN_HIRESERVE** are reserved for special values — they do not refer to the section header table.
SHN_LOPROC	0xff00	Section indices between **SHN_LOPROC** and **SHN_HIPROC** are reserved for processor-specific values.
SHN_MIPS_ACOMMON	0xff00	Allocated common symbols in a DSO[a,e].
SHN_MIPS_TEXT	0xff01	Reserved (obsolete).

SiliconGraphics Computer Systems

Table 15 *st_shndx* Field Special Values

Name	Value	Semantics
SHN_MIPS_DATA	`0xff02`	Reserved (obsolete).
SHN_MIPS_SCOMMON	`0xff03`	*gp*-addressable common symbols[b] (relocatable objects only).
SHN_MIPS_SUNDEFINED	`0xff04`	*gp*-addressable undefined symbols[c] (relocatable objects only).
SHN_MIPS_LCOMMON	`0xff05`	Local common, equivalent to **SHN_COMMON**, except that the common block will be allocated in a local section, i.e. one replicated for each process in a multi-process program sharing memory (see **SHF_MIPS_LOCAL** in Section 2.3).[d,e]
SHN_MIPS_LUNDEFINED	`0xff06`	Local undefined symbol, equivalent to **SHN_UNDEFINED**, except that the symbol must resolve to a local section, i.e. one replicated for each process in a multi-process program sharing memory (see **SHF_MIPS_LOCAL** in Section 2.3).[d]
SHN_HIPROC	`0xff1f`	Section indices between **SHN_LOPROC** and **SHN_HIPROC** are reserved for processor-specific values.
SHN_ABS	`0xfff1`	The symbol's value is absolute and does not change due to relocation.
SHN_COMMON	`0xfff2`	The symbol labels a common block which has not yet been allocated[a]. Its *st_value* specifies alignment, similar to a section header's *sh_addralign* field. Its size is the number of bytes required.
SHN_HIRESERVE	`0xffff`	Section indices between **SHN_LORESERVE** and **SHN_HIRESERVE** are reserved for special values — they do not refer to the section header table.

[a] Normal **SHN_COMMON** symbols are not allocated in DSOs, which means that if they are not allocated elsewhere (in the main executable or another DSO), ***rld*** must allocate them at runtime. **SHN_MIPS_ACOMMON** symbols are used in DSOs to mark common that has been allocated in advance to avoid runtime allocation, but still have common semantics, i.e. are not initialized and may be preempted by non-common definitions from any DSO. The *st_value* of such a symbol is its virtual address. It may be relocated, but the alignment of its address must be preserved up to modulo 65,536.

[b] **SHN_MIPS_SCOMMON** symbols are common symbols which must be allocated within 32,768 bytes of *gp*.

[c] **SHN_MIPS_SUNDEFINED** symbols are undefined symbols which must be resolved to addresses within 32,768 bytes of *gp*.

[d] **SHN_MIPS_LCOMMON** and **SHN_MIPS_LUNDEFINED** symbols must be consistently defined in a program, i.e. every appearance of the symbol must be either an undefined **SHN_MIPS_LCOMMON** reference, a **SHN_MIPS_LUNDEFINED** reference, or a definition in a **SHF_MIPS_LOCAL** section. These symbols may not be gp-relative.

[e] **SHN_MIPS_ACOMMON** symbols with values (virtual addresses) in **SHF_MIPS_LOCAL** sections are equivalent to **SHN_MIPS_LCOMMON** symbols, except that they are pre-allocated by the static linker.

In executable and shared object files, the symbol's value is a virtual address (if defined), and the section header index is irrelevant. If the section header index of a function symbol is **SHN_UNDEF** and the *st_value* is non-zero, it is the virtual address of a stub for lazy evaluation of a symbol defined in one of the associated DSOs.

ELF-32

ELF-32: The extension of using *st_other* for specifying DSO-related scope attributes is used in ELF-32 beginning with the IRIX 5.1 release for linker-generated symbols which must be protected. As this field is currently unused, there should be no compatibility issues unless there are tools which attempt to enforce non-use. Tools which ignore this field will be unable to cope with the implied symbol resolution.

2.5.1 Split Common Symbols

It is sometimes desirable for the compiler to split a Fortran COMMON block into multiple pieces for separate allocation (e.g. to control relative position in the processor cache). The MIPSpro compilers do so under appropriate options, and represent the resulting decomposition as described below. The static linker (ld) then determines whether the decomposition is safe, and reassembles the COMMON if not.

A split common symbol is represented in a relocatable object by a normal symbol for the original COMMON block, plus a split common symbol for each element of the decomposition. The split common symbol inherits its alignment, binding type, export class, and section index field information from the parent common, allowing its symbol table entry to be redefined as given by the modified version of Table 11 in Table 16 below:

Table 16 ELF-64 Split Common Component Symbol Table Element

Name	Type	Comments
st_name	**Elf64_Word**	Component name's index into string table
st_info	**Elf64_Byte**	**STB_SPLIT_COMMON**
st_other	**Elf64_Byte**	**STO_SC_ALIGN_UNUSED**
st_shndx	**Elf64_Section**	Symbol index of parent common symbol
st_value	**Elf64_Addr**	Offset of component from base of parent common symbol
st_size	**Elf64_Xword**	Component size

A split common symbol is identified by the **STB_SPLIT_COMMON** value of the *st_info* field, requiring the alternate interpretation of its other fields given above.

Compiler Processing

Compiler Requirements:

- The component symbols produced by the compiler must be a partition of the parent COMMON. That is, the first should start at offset zero, the second at offset size(first), etc., and length(parent) = offset(last)+size(last) = sum(sizes).
- Subject to the above requirement, the compiler is allowed to split common blocks arbitrarily, and the linker is responsible for identifying problems and reassembling the original common block.

Linker Processing

Linker Requirements:

- The linker must produce a final partition which is consistent with each of the individual objects' partitions in the sense that each component from an input relocatable object falls entirely within one of the output components.
- If the linker partitions the common, it must replace the component symbols by normal (e.g. common, bss, or data) symbols, producing an object which contains no split common component symbols (and therefore looks like an ABI-compliant object). It may do so either by producing a distinct regular symbol for each component, or by replacing the component symbols by appropriate offsets from a single symbol for the common. Similarly, if it reassembles the common, it must remove the component symbols.
- If the linker partitions the common, the resulting symbols must not be exported, to avoid inadvertent incorrect references if one of the referenced DSOs is replaced later with a version which references the name.
- The following situations require that the linker reassemble the original common:
  a. There is an explicit linker request to export the common block symbol (implying that some DSO may contain a reference to it with unknown assumptions).
  b. The linker finds a relocation against the common block symbol (implying code generated to reference the original common block with unknown assumptions).

c. The linker finds a reference to the common block symbol from any DSO (implying that the DSO contains a reference to it with unknown assumptions).

d. The common block symbol is initialized with a single BLOCK DATA (i.e. it is defined at a specific location in a data section).

e. The linker finds a PU or object file in which the common has not been split (a variant of (a) and (c)).

A linker implementation may choose to reassemble split common blocks in other circumstances.

2.6 Hash Table

A hash table section is unchanged from [ABI32]. It has the following attributes:

name	**.hash**
sh_type	**SHT_HASH**
sh_link	Section header index of the associated symbol table
sh_info	0
sh_flags	**SHF_ALLOC**
requirements	may not be stripped

See [ABI32] for a description of the hash table structure and hash function, which are unchanged.

2.7 Register Information Section

This section is strictly a 32-bit ABI section, which specifies the register usage of the code in an object file. In 64-bit ELF, this section is obsolete, and is superceded by the Options Section described in Section 2.8 below. Its section attributes are:

name	**.reginfo**
sh_type	**SHT_MIPS_REGINFO**
sh_link	**SHN_UNDEF**
sh_info	0
sh_flags	none

requirements	obsolete — not strippable if present in ELF-32 file

The structure of a Register Information descriptor is given by Table 19 in Section 2.8 below.

ELF-32

ELF-32: MIPS tools will ignore this section in 32-bit ELF if a **.MIPS.options** section is present.

2.8 Options Section

This section specifies miscellaneous options to be applied to an object file. An options section is required, and must contain at least an **ODK_REGINFO** descriptor (below). In a shared executable or DSO, the options section should immediately follow the program header table for best startup performance, and the **.dynamic** section must contain a **DT_MIPS_OPTIONS** tag pointing to it. In a non-shared program (which is not ABI-conformant), the options section must immediately follow the program header table.

Its section attributes are:

name	**.MIPS.options**
sh_type	**SHT_MIPS_OPTIONS**
sh_link	**SHN_UNDEF**
sh_info	0
sh_flags	**SHF_ALLOC + SHF_MIPS_NOSTRIP**
requirements	mandatory, may not be stripped

An options record consists of a sequence of variable length (and variable format) descriptors, which may apply either to the entire object file or to a specific section. Each such descriptor begins with the following header:

Table 17 Options Descriptor Header (*Elf_Options*)

Field Name	Type	Comments
kind	**Elf64_Byte**	Determines interpretation of variable part of descriptor — see Table 18 below

Table 17

Options Descriptor Header (*Elf_Options*)

Field Name	Type	Comments
size	**Elf64_Byte**	Byte size of descriptor, including this header [a]
section	**Elf64_Section**	Section header index of section affected, or 0 for global options
info	**Elf64_Word**	Kind-specific information

[a] Descriptors may have arbitrary size (up to 255 bytes). However, they must be 8-byte aligned, and must be null padded if the given size is not 0 mod 8.

The Options Descriptor kinds are given in Table 18 below. The various descriptors associated with them follow. For each descriptor kind, the associated table gives the value of the *size* field, the usage of the *info* field, and any additional fields required.

Table 18

Options Descriptor Kinds

Constant Name	Value	Comments
ODK_NULL	0	Undefined
ODK_REGINFO	1	Register usage information
ODK_EXCEPTIONS	2	Exception processing options
ODK_PAD	3	Section padding options
ODK_HWPATCH	4	Hardware patches applied
ODK_FILL	5	Linker fill value
ODK_TAGS	6	Space for tool identification
ODK_HWAND	7	Hardware AND patches applied
ODK_HWOR	8	Hardware OR patches applied
ODK_GP_GROUP	9	GP group to use for text/data sections
ODK_IDENT	10	ID information
ODK_PAGESIZE	11	Page size information

The **.MIPS.options** section replaced **.reginfo** in ELF-32 with the MIPSpro 6.0 compilers, although it remains in use in the 5.x (ucode) compilers.

2.8.1 Register Information Option Descriptor

The **ODK_REGINFO** descriptor supercedes what used to be in the special .reginfo section. The structure of a Register Information descriptor is given by Table 19 below.

Table 19 Register Information Structure

Field Name	Type	Comments
kind	**Elf64_Byte**	value is **ODK_REGINFO**
size	**Elf64_Byte**	value is 40
section	**Elf64_Section**	0
info	**Elf64_Word**	unused
ri_gprmask	**Elf64_Word**	Mask of general registers used
ri_pad	**Elf64_Word**	Unused padding field (for alignment of following fields)
ri_cprmask[4]	**Elf64_Word[4]**	Mask of coprocessor registers used
ri_gp_value	**Elf64_Addr**	Initial value of **gp** [a]

[a] If there is no register information descriptor, the initial value of **gp** is assumed to be 0. The significance of this value is that, for any **SHT_MIPS_GPREL** section, if its start address (as given by the section's **sh_addr**) is specified as **x** (which will usually be zero), then gp is assumed to be initialized to its relocated start address plus (*ri_gp_value*-**x**). The initial addends of any **R_MIPS_GPREL**-relocated values will be correct offsets if the section is not moved relative to **gp**.

ELF-32

ELF-32: We will use this section instead of **.reginfo** beginning with IRIX 6.0, since we require the additional descriptors described below. However, **.reginfo** may still be generated for the benefit of foreign (and back-rev) tools.

2.8.2 Exception Information Option Descriptor

The **ODK_EXCEPTIONS** descriptor contains information only in the *info* field of the basic options descriptor header, as given by the masks in Table 20 below. Its *size* is 8 bytes.

Table 20 Exception Information *info* Field Masks

Mask Name	Value	Comments
OEX_FPU_MIN	`0x0000001f`	Min FPU exception enable [a]
OEX_FPU_MAX	`0x00001f00`	Max FPU exception enable [a]
OEX_PAGE0	`0x00010000`	Page zero of the virtual address space must be mapped [b]
OEX_SMM	`0x00020000`	Run in sequential memory mode [c]
OEX_PRECISEFP	`0x00040000`	Run in precise FP exception mode [d]
OEX_DISMISS	`0x00080000`	Dismiss invalid address traps [e]

[a] These masks bound the setting of the FPU exception enable masks at runtime. The runtime mask may enable only bits in the maximum mask, and must enable bits in the minimum mask. See discussion below.

[b] If set, loads from page zero of the virtual address space must not cause invalid address faults. However, page zero may be write-protected. "Page zero" here implies the minimum, given by **ELF_MIPS_MINPGSZ** in ***elf.h***.

[c] If set, and the process is running on an R8000 or other processor with a sequential memory mode, execute in that mode.

[d] If set, and the process is running on an R8000 or other processor with a distinct mode for precise floating point exceptions, execute in that mode.

[e] If set, any invalid address traps encountered should be dismissed without aborting or otherwise notifying the running process.

Linker Processing

Linkage and Execution: The static linker (***ld***) must combine the descriptors from all object files linked as follows. The **OEX_FPU_MIN** fields are OR'ed together, so that any exception enable required by any of the objects will be set for the process. Similarly, the **OEX_FPU_MAX** fields are AND'ed together, so that any exception required to be suppressed by any of the objects will be suppressed for the process. The **OEX_PAGE0**, **OEX_SMM**, **OEX_PRECISEFP**, and **OEX_DISMISS** flags are OR'ed together. The linker should always produce an **ODK_EXCEPTIONS** descriptor even if none of the linked objects contained one, so that simple tools can be used to manipulate these options. In such a case, the *info* field should contain the value **OEX_FPU_MAX**, i.e. any FP exceptions allowed, and no other options set.

At execution time, the dynamic linker (***rld***) must perform a similar task to combine the descriptors from the main executable and any DSOs, and it must perform the appropriate actions to achieve the implied state (typically making calls to the kernel). In addition, it must retain the implied state for reference — in the event that a **dlopen** call is made to open a new DSO, its state must be checked for compatibility with the current state, and that state adjusted as required. The runtime linker is not required to back out the requirements of a DSO which is subsequently removed from the process image via **dlclose**.

For a non-shared program, the kernel or the runtime system (e.g. crt0) must perform the same task, setting the implied initial state in the running process.

Both the static and dynamic linkers should report incompatible requirements of their components, i.e. an exception enable bit which is set in the minimum mask and unset in the maximum mask, as errors. The runtime system is not required to enforce retention of the specified modes in the face of explicit attempts to set them by the running process, but it may do so.

ELF-32

ELF-32: MIPS generates and uses this descriptor beginning with IRIX 6.0. It may be necessary to suppress it for pure-ABI objects.

2.8.3 Section Padding Option Descriptor

The ODK_PAD descriptor specifies padding required for the referenced data section. The linker must provide for at least the indicated number of bytes preceding or following the data section to be valid parts of the virtual address space, guaranteed not to cause invalid address faults. This facility is intended to allow the code generator to produce memory references which may be beyond the referenced data object (e.g. for software pipelining), with the assurance that they will not cause memory faults at runtime.

If the writable flags are not set, the linker may provide padding simply by arranging for other data sections to be contiguous to the section specified, those other data sections need not be writable. If the writable flags are set, writable empty space must be provided. If padding must be applied to a symbol (e.g. because it is undefined or COMMON, and its ultimate section is unknown), the 16-bit *section* is a symbol table section, and the *pad_symindex* field specifies a 32-bit symbol index. In this case, since the symbol may be allocated by another object file in the midst of a larger section, the writable flags may not be set.

The format of this descriptor is given in Table 21 below:

Table 21 Section Padding Descriptor

Field Name	Type	Comments
kind	**Elf64_Byte**	value is **ODK_PAD**
size	**Elf64_Byte**	value is 16
section	**Elf64_Section**	Section to be padded [a]
info & 0x0001	**mask**	Prefix writable (**OPAD_PREFIX**)
info & 0x0002	**mask**	Postfix writable (**OPAD_POSTFIX**)
info & 0x0004	**mask**	Pad symbol [b] (**OPAD_SYMBOL**)
pad_prefix_size	**Elf64_Half**	Size (bytes) of prefix required
pad_postfix_size	**Elf64_Half**	Size (bytes) of postfix required
pad_symindex	**Elf64_Word**	Symbol index if **OPAD_SYMBOL** is set

[a] The section field normally references a data section to be padded or, if **OPAD_SYMBOL** is set, a symbol table section in which to find a symbol to be padded. If it is zero, the descriptor applies to all data sections, and the writable flags may not be set in this case.

[b] If this flag is set, the section field is a symbol table section, and the *pad_symindex* field specifies a 32-bit symbol index instead of a section for padding (generally undefined or COMMON). The writable flags may not be set in this case.

ELF-32

ELF-32: We generate and use this descriptor beginning with IRIX 6.0. It may be necessary to suppress it for pure-ABI objects.

Linker Processing

Linkage: It is technically possible for the static linker (***ld***) to postpone processing of padding information (e.g. for symbols and/or sections allocated at the beginning or end of a segment, or for unallocated COMMON symbols), leaving the dynamic linker (***rld***) to process residual padding descriptors. However, the MIPSpro static linker currently processes padding completely, dealing with unallocated COMMON symbols by turning them into ACOMMON symbols with proper padding, and the MIPSpro dynamic linker does not deal with padding.

2.8.4 Hardware Patch Option Descriptor

The **ODK_HWPATCH** descriptor contains flags indicating whether various patches required for specific hardware platforms have been applied to the executable or DSO. It contains information only in the *info* field of the basic options descriptor header, as given by the masks in Table 22 below. Its *size* is 8 bytes. This descriptor is required in all object files, to allow simple post-generation patching.

Table 22 Hardware Patch Options Descriptor

Mask Name	Value	Comments
OHW_R4KEOP	`0x00000001`	Patch for R4000 branch at end-of-page bug
OHW_R8KPFETCH	`0x00000002`	Object contains prefetch instructions which may cause R8000 prefetch bug to occur
OHW_R5KEOP	`0x00000004`	Patch for R5000 branch at end-of-page bug
OHW_R5KCVTL	`0x00000008`	R5000 cvt.[ds].l bug: clean=1
OHW_R10KLDL	`0x00000010`	Requires patch for R10000 misaligned load.

Linker Processing

The static linker must merge (inclusive OR) the **OHW_R8KPFETCH** flags from each of the objects it links. The **OHW_R4KEOP**R5000 cvt.[ds].l bug. clean=1 and **OHW_R5KEOP** flags are added by tools after linking.

2.8.5 Hardware AND/OR Patch Option Descriptors

The **ODK_HWAND** / **ODK_HWOR** descriptors, like **ODK_HWPATCH**, contain flags indicating whether various patches required for specific hardware platforms have been applied to the object file. They contain information in the *info* field of the basic options descriptor header, plus the following 8 bytes. Their *size* is 16 bytes. These descriptors are required in all object files, to allow simple post-generation patching. The descriptor layout is given in Table 23 below.

Table 23 Hardware AND/OR Patch Option Descriptor Structure

Field Name	Type	Comments
kind	**Elf64_Byte**	value **ODK_HWAND** or **ODK_HWOR**
size	**Elf64_Byte**	value is 16
section	**Elf64_Section**	0
info	**Elf64_Word**	32 flags: see Table 24

Table 23 Hardware AND/OR Patch Option Descriptor Structure

Field Name	Type	Comments
hwp_flags1	**Elf64_Word**	32 flags: see Table 24
hwp_flags2	**Elf64_Word**	32 flags: see Table 24

Table 24 Hardware Patch AND/OR Options Descriptor Flags

Mask Name	Value	Comments
ODK_HWAND *info* masks:		
OHWA0_R4KEOP_CHECKED	`0x00000001`	Object checked for R4K end-of-page bug.
OHWA0_R4KEOP_CLEAN	`0x00000002`	Object verified clean of R4K end-of-page bug.
ODK_HWAND *hwp_flags1* masks:		
OHWA1_...	`0x????????`	None yet defined.
ODK_HWAND *hwp_flags2* masks:		
OHWA2_...	`0x????????`	None yet defined.
ODK_HWOR *info* masks:		
OHWO0_FIXADE	`0x00000001`	Object requires call to fixade
ODK_HWOR *hwp_flags1* masks:		
OHWO1_...	`0x????????`	None yet defined.
ODK_HWAND *hwp_flags2* masks:		
OHWO2_...	`0x????????`	None yet defined.

Linker Processing

The static linker must merge the **ODK_HWAND** (bitwise AND) and **ODK_HWOR** (bitwise inclusive OR) flags from each of the objects it links. Generating tools should initialize the flags to zero for fields they do not understand, and the linker should assume that missing descriptors contain zeroes.

2.8.6 Fill Value Option Descriptor

The **ODK_FILL** descriptor contains information only in the *info* field of the basic options descriptor header, specifically the value used by the linker to fill uninitialized space. Its *size* is 8 bytes.

2.8.7 Tags Option Descriptor

The **ODK_TAGS** descriptor initially contains only zero-filled space (40 bytes). It is reserved for tools to identify processing that has occurred. Its *size* is 48 bytes.

Table 25 Tags Option Descriptor

Field Name	Type	Comments
kind	**Elf64_Byte**	value is **ODK_TAGS**
size	**Elf64_Byte**	value is 48
section	**Elf64_Section**	0 (unused)
info	**mask**	0 (unused)
tags	**Elf64_Byte[40]**	initially 0. Bytes currently reserved: 0..4 (Desktop) 32..39 (other vendors)

The purpose of this descriptor is to allow tools to mark the object (executable or DSO) without substantially rewriting the file, as would be required to add a new descriptor or section. Space in this descriptor should be allocated very carefully, preferably a byte at a time. The **.note** section is more appropriate if more space is required. The intent is that this allocation last essentially forever. Although expansion is theoretically possible, doing so would eliminate its benefit for files created before the expansion.

The last 8 bytes of the *tags* field are reserved for vendor-specific use by non-MIPS/SGI vendors. Note, however, that such usage may conflict with other vendors' usage, and should therefore be limited to files which is not expected to be handled by other vendors' software.

2.8.8 GP Group Option Descriptor

The **ODK_GP_GROUP** descriptor is used to specify to which GP group text and data sections are to be assigned, when this is determined prior to linking (e.g. by interprocedural analysis). Its format is:

Table 26 GP Group Option Descriptor

Field Name	Type	Comments
kind	**Elf64_Byte**	value is **ODK_GP_GROUP**
size	**Elf64_Byte**	value is total size (8 + 2*section count)
section	**Elf64_Section**	0 (unused)
info & 0x0000ffff	**mask**	*OGP_GROUP*: GP group number
info & 0x00010000	**mask**	*OGP_SELF*: GP group is self-contained
section_ids	**Elf64_Section [size/2-4]**	Section IDs for those sections with the given GP group number

The number of sections allocated by a single such descriptor is limited by the descriptor size limit of 255 bytes to 123. Multiple descriptors may be used if more sections need to be included. In such a case, the **OGP_SELF** flag must match for all occurrences of the same group number.

If this descriptor is present, it must be preceded by an **ODK_IDENT** descriptor (below) unless all the GP groups are self-contained, and the **EF_MIPS_OPTIONS_FIRST** flag must be set in the ELF header.

Linker Processing

Each section identified by ID in the *section_ids* list must be allocated by the linker to the same GP group as all other sections with the same GP group number (but the linker may merge multiple groups if they fit). A section without such an entry may be allocated arbitrarily. If the *OGP_SELF* flag is set for the group, this requirement is restricted to sections in the file containing this **ODK_GP_GROUP** descriptor. Otherwise, this requirement applies to sections with this *OGP_GROUP* number in all files containing **ODK_IDENT** descriptors with the same *identifier* field value. (See the description of the **ODK_IDENT** descriptor below for further information.)

2.8.9 Ident Option Descriptor

The **ODK_IDENT** descriptor is used to provide identification information in support of cross-file features. Its format is:

Table 27 Ident Option Descriptor

Field Name	Type	Comments
kind	**Elf64_Byte**	value is **ODK_IDENT**
size	**Elf64_Byte**	value is 16
section	**Elf64_Section**	0 (unused)
info & 0x0000ffff	**mask**	*OGP_GROUP*: default GP group number
info & 0x00010000	**mask**	*OGP_SELF*: default GP group self-contained
identifier	**Elf64_XWord**	Timestamp or similar identifier

If this descriptor is present, the **EF_MIPS_OPTIONS_FIRST** flag must be set in the ELF header.

The *OGP_GROUP* field provides a GP group number for all sections which do not appear in an **ODK_GP_GROUP** descriptor. The *OGP_SELF* flag applies to that group, as described with the **ODK_GP_GROUP** descriptor above.

All object files using GP groups in **ODK_IDENT** or **ODK_GP_GROUP** descriptors with the *OGP_SELF* flag not set must have the same non-zero *identifier* field value. Any object file which specifies GP groups for some of its sections using the **ODK_GP_GROUP** descriptor (above) must contain one of these descriptors (preceding the **ODK_GP_GROUP** descriptor) unless its GP groups are all self-contained (i.e. have their *OGP_SELF* flags set, and need not be combined with identically number GP groups in another object).

Linker Processing

A typical scenario would have IPA specify the same non-zero *identifier* field for each of the files it processes together, and an ***ld -r*** command set the *OGP_SELF* flag after verifying that all the *identifiers* match and combining the various contributions to the same GP group. Another scenario would involve the compiler itself deciding to assign multiple GP groups to an object, in which case it would set *OGP_SELF* and optionally assign a non-zero *identifier*. It is a link-time error to encounter multiple object files with **ODK_GP_GROUP** descriptors, and different *identifier* fields with *OGP_SELF* not set in their **ODK_IDENT** descriptors.

2.8.10 Page Size Option Descriptor

The **ODK_PAGESIZE** descriptor is used to specify page sizes to be used in running a program. *This is currently an unimplemented proposal.* Its format is:

Table 28 Page Size Option Descriptor

Field Name	Type	Comments
kind	**Elf64_Byte**	value is **ODK_PAGESIZE**
size	**Elf64_Byte**	value is 8
section	**Elf64_Section**	0 (unused)
info & 0x000000ff	**mask**	*OPS_DATA*: data page size to use
info & 0x0000ff00	**mask**	*OPS_STACK*: stack page size to use
info & 0x00ff0000	**mask**	*OPS_TEXT*: text page size to use

This descriptor specifies system page sizes to be used in running the containing object as part of a process. Each of the three one-byte values is interpreted as a power-of-two exponent giving the desired page size, with the value zero meaning that the default is to be used, and the value one meaning that the runtime environment variables **PAGESIZE_DATA**, **PAGESIZE_STACK**, **PAGESIZE_TEXT,** or **PAGESIZE_ALL** are to be queried. The data page size applies to both static data segments and the heap.

This descriptor is optional, and may be ignored by a system which does not support per-process variable page sizes. A system may or may not recognize it in relocatable object files. The rules for combining conflicting values in relocatable object files (if recognized), or in multiple DSOs comprising a program, are implementation defined.

2.9 Relocation

Any section may have an associated **SHT_REL** and/or **SHT_RELA** section, containing relocation operations for objects in the section. Its section attributes are:

name	**.rel***name or* **.rela***name* (where *name* is the relocated section)
sh_type	**SHT_REL** *or* **SHT_RELA**
sh_link	Section header index of the associated symbol table
sh_info	Section header index of the section to be relocated

sh_flags	None by default
requirements	object file relocation sections which are processed by ***ld*** need not be transferred to executable/DSO, or may be strippable if placed there.

The content of each section is an array of relocation records, as described by Table 29 below:

Table 29 Relocation Operation (*Elf64_Rel, Elf64_Rela*)

Field Name	Type	Comments
r_offset	**Elf64_Addr**	Where to apply relocation: relocatable: byte offset in section executable: virtual address
r_sym	**Elf64_Word**	Symbol index
r_ssym	**Elf64_Byte**	Special symbol — see Table 30
r_type3	**Elf64_Byte**	Relocation type — see Table 32
r_type2	**Elf64_Byte**	Relocation type — see Table 32
r_type	**Elf64_Byte**	Relocation type — see Table 32
r_addend	**Elf64_Sxword**	Explicit addend for relocation operation (*Elf64_Rela* only)

2.9.1 Rules for Interpreting Relocation Records

The relocation operation consists of applying the operation(s) implied by the type subfield(s) to operands which may include the address or offset of the storage unit being relocated (*r_offset*), the current content of the storage unit being relocated, the value of the symbol indexed by *r_symndx*, a special symbol value (*r_ssym*) and/or the addend (*r_addend*). If the symbol index is **STN_UNDEF** (0), it is treated as having value 0.

A number of relocation operations may be applied to a single address. If they are consecutive in the relocation section, they are interpreted according to the following rules:

a1. The first operation takes its addend from either the r_addend field of the relocation operation record (if it is of type **Elf64_Rela**), or from the location to be relocated, as implied by the operation (if the relocation record is of type **Elf64_Rel**).

a2. Each subsequent operation takes as its addend the result of the previous operation. All such intermediate results, and all relocation arithmetic, are in the natural pointer length of the object, i.e. 64 bits for ELF-64, and 32 bits for ELF-32.

a3. Only the final operation actually modifies the location relocated.

o1. Up to three operations may be specified per record, by the fields *r_type*, *r_type2*, and *r_type3*. They are applied in that order, and a zero field implies no further operations *from this record*. (The following record may continue the sequence if it references the same offset.)

s1. The first operation in a record which references a symbol uses the symbol implied by *r_sym.*

s2. The next operation in a record which references a symbol uses the special symbol value given by the *r_ssym* field, as described in Table 30.

s3. A third operation in a record which references a symbol will assume a NULL symbol, i.e. value zero. This is useful for operations which do nothing but insert the relocated value into the proper instruction field.

The implication of the rules (a1)-(a3) is that a relocation type consists of two components. The first component is the operation to be performed, which is always relevant (although many of the relocation types will have no effect given a NULL symbol). The second component is a specification of the field to be relocated, which is relevant only for the first operation in the sequence (where it may specify the addend for an **Elf64_Rel** relocation) and for the last operation in a sequence (where it specifies the field rewritten by the relocation). This is compatible with old-style MIPS relocations if one assumes that every relocation sequence has exactly one element (which was probably true, except perhaps in cases where the new definition won't affect the result).

ELF-32

The purpose of this more complex definition is to allow us to specify more complex relocations by composing simple relocations instead of requiring that we always define additional relocation types. This will generalize our relocation capabilities significantly without many new operations. The new composition rules are observed for 32-bit object files as well in the IRIX 6.0 linker and beyond (except for those related to having multiple relocation types in a record).

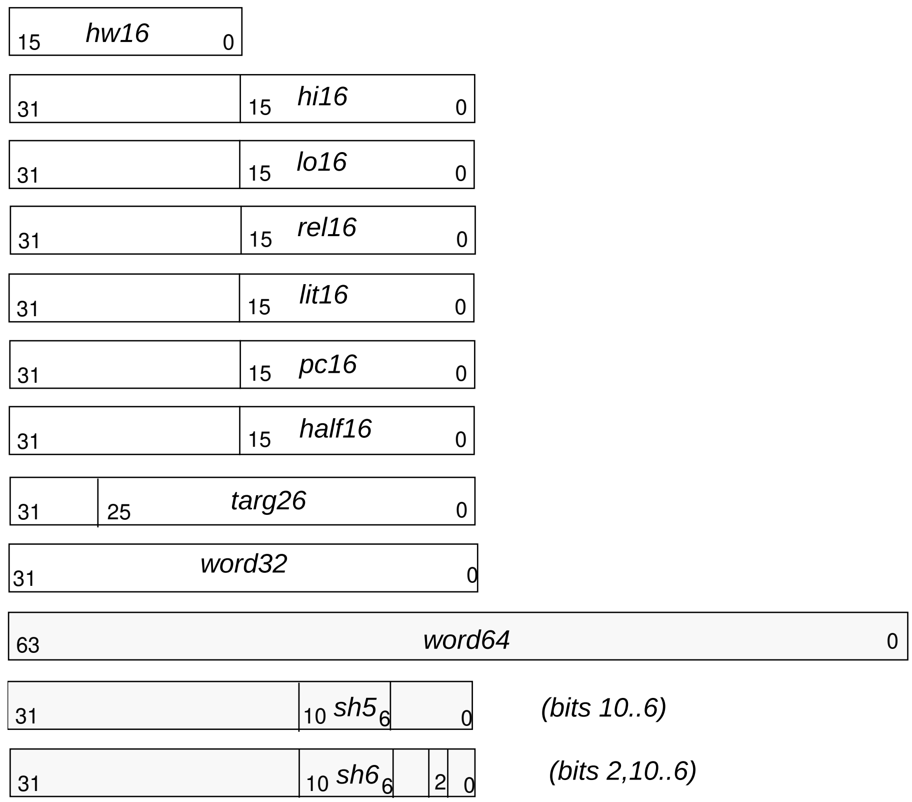
Table 30 Special Relocation Symbols

Name	Value	Description
RSS_UNDEF	0	None — value is zero.
RSS_GP	1	Value of **gp**
RSS_GP0	2	Value of **gp** used to create object being relocated
RSS_LOC	3	Address of location being relocated

2.9.2 Semantics of the Relocation Types

Relocation may be applied to the following fields of an instruction or data word. Note that regardless of the position of a relocated field within an instruction, the offset/address specified in *r_offset* is that of the full instruction, not of the field. Also note that the instruction or datum being relocated need not be aligned.

Figure 1 Relocatable Fields



In the lists of relocation operations, the operation is described using operands from Table 31 below.

Table 31 Relocation Operands

Operand	Description
A	Represents an addend obtained as the value of the field being relocated prior to relocation (**.rel**), from a **.rela** addend field, or as the preceding result for a composed relocation (either).

Table 31 Relocation Operands

Operand	Description
AHL	An address addend formed as follows. In **.rela** sections, it is identical to an A addend. In a **.rel** section, a pair of adjacent relocations, one a *hi16* and the other a *lo16*, each provide a 16-bit partial addend. The *hi16* halfword is shifted left 16 bits, the *lo16* halfword is sign extended, and the two resulting values are added. (The two relocations need not actually be adjacent in a **.rel** section -- a single *hi16* addend may be used with multiple *lo16* addends -- but processing this combination requires fallable heuristics, so these relocations should not be used in **.rel** sections.)
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, unless the symbol is STB_LOCAL of type STT_SECTION, in which case S represents the final *sh_addr* minus the original *sh_addr*.
G	The offset into the global offset table at which the address of the relocation entry symbol, adjusted by the addend, resides during execution.
GP	The final gp value to be used for the relocatable, executable, or DSO being produced.
GP0	The gp value used to create the relocatable object. See Table 19.
EA	The effective address of the symbol prior to relocation.
L	The mapping table offset of a merged section, e.g. .lit4. Prior to relocation, the addend field (in the instruction) contains an offset into the object’s global data area. During relocation, the sections are merged, removing duplicate entries, and a mapping table is constructed to map the original offsets to the new offsets.

The actual relocation operations supported are described below in Table 32. The name and value columns are the relocation type, which is one of the *r_type** fields. The field column specifies the affected field of the storage unit being relocated (only for the last operation in a composed relocation sequence). The "T_" prefix indicates that excess high-order bits are to be truncated; the "V_" prefix indicates that the value is verified to fit in the field, with an error generated if it does not. The symbol column specifies the kind of symbol to which the description applies.

Any of the relocation types may appear in either a **SHT_REL** or a **SHT_RELA** relocation section, except that relocation types involving

AHL operands are forbidden in a 64-bit **SHT_REL** section and discouraged in a 32-bit **SHT_REL** section. In the latter case, they WILL NOT BE SUPPORTED unless the ordering constraints imposed by table footnote (b) are observed. Also note that some relocations (e.g. **R_MIPS_HIGHER**, **R_MIPS_HIGHEST**) will normally be impossible to specify in a **SHT_REL** section unless the required addend is small. An **SHT_RELA** section must also be used for such relocations if the required addend could become too large for its field in an *ld -r* partial link, even if the value is small as generated by the original object file producer.

Several of the original MIPS relocation types may be used only for the operation implied, and not for the field specified, in a multi-operation sequence as described above. The name fields for those types are shaded, and some are given alternate names in the table below to emphasize the alternate interpretation.

Table 32 Relocation Types

Name	Value	Field	Symbol	Calculation
R_MIPS_NONE	0	none	n/a	none
R_MIPS_16	1	V-half16	any	S + sign_extend(A)
R_MIPS_32 R_MIPS_ADD	2	T-word32	any	S + A
R_MIPS_REL32 R_MIPS_REL	3	T-word32	any	S + A - EA
R_MIPS_26	4	T-targ26	local [a]	(((A << 2) | (P&0xf0000000)) + S) >> 2
			external [a]	(sign_extend(A<<2) + S) >> 2
R_MIPS_HI16 [b, c]	5	T-hi16	any	%high (AHL + S) [d]
R_MIPS_LO16 [b, c]	6	T-lo16	any	AHL + S
R_MIPS_GPREL16 R_MIPS_GPREL	7	V-rel16	external	sign_extend(A) + S - GP
			local	sign_extend(A) + S + GP0 - GP
R_MIPS_LITERAL	8	V-lit16	local	sign_extend(A) + L

Table 32 Relocation Types

Name	Value	Field	Symbol	Calculation
R_MIPS_GOT16 [e] R_MIPS_GOT	9	V-rel16	external	G
			local	[f]
R_MIPS_PC16	10	V-pc16	external	sign_extend(A) + S - P
R_MIPS_CALL16 [e,m] R_MIPS_CALL[m]	11	V-rel16	external	G
R_MIPS_GPREL32	12	T-word32	local	A + S + GP0 - GP
R_MIPS_SHIFT5	16	V-sh5	any	S
R_MIPS_SHIFT6	17	V-sh6	any	S
R_MIPS_64 [g]	18	T-word64	any	S + A
R_MIPS_GOT_DISP	19	V-rel16	any	G
R_MIPS_GOT_PAGE	20	V-rel16	any	[h]
R_MIPS_GOT_OFST	21	V-rel16	any	[h]
R_MIPS_GOT_HI16	22	T-hi16	any	%high(G)[d]
R_MIPS_GOT_LO16	23	T-lo16	any	G
R_MIPS_SUB	24	T-word64	any	S - A
R_MIPS_INSERT_A	25	T-word32	any	Insert addend as instruction immediately prior to addressed location. [i]
R_MIPS_INSERT_B	26	T-word32	any	
R_MIPS_DELETE	27	T-word32	any	Remove the addressed 32-bit object (normally an instruction). [j]
R_MIPS_HIGHER	28	T-hi16	any	%higher(A+S)[k]
R_MIPS_HIGHEST	29	T-hi16	any	%highest(A+S)[l]
R_MIPS_CALL_HI16[m]	30	T-hi16	any	%high(G)[d]
R_MIPS_CALL_LO16[m]	31	T-lo16	any	G
R_MIPS_SCN_DISP	32	T-word32	any	[n] S+A-scn_addr (Section displacement)
R_MIPS_REL16	33	V-hw16	any	S + A

Table 32 Relocation Types

Name	Value	Field	Symbol	Calculation
R_MIPS_ADD_IMMEDIATE	34	V-half16	any	[o]S + sign_extend(A)
R_MIPS_PJUMP	35	T-word32	any	Deprecated (protected jump)
R_MIPS_RELGOT	36	T-word32	any	[q]S + A - EA
R_MIPS_JALR	37	T-word32	any	[p]Protected jump conversion

Notes:

[a] A *local* symbol is one with binding **STB_LOCAL** and type **STT_SECTION**. Otherwise, a symbol is *external*.

[b] An **R_MIPS_HI16** must be followed immediately by an **R_MIPS_LO16** relocation record in a **SHT_REL** section. The contents of the two fields to be relocated are combined to form a full 32-bit addend AHL. An **R_MIPS_LO16** entry which does not immediately follow a **R_MIPS_HI16** is combined with the most recent one encountered, i.e. multiple **R_MIPS_LO16** entries may be associated with a single **R_MIPS_HI16**. Use of these relocation types in a **SHT_REL** section is discouraged and may be forbidden to avoid this complication.

[c] The special symbol name _gp_disp, used for relocating the calculation of gp on entry to a DSO in 32-bit files, is not supported in ELF-64 or in the new 32-bit ABI. Instead, these relocations should be composed with **R_MIPS_GPREL** applied to an explicit symbol for the entry point of the subprogram. See the examples below.

[d] The %high(x) function is (x - (short)x) >> 16.

[e] The first instance of an **R_MIPS_GOT*** or an **R_MIPS_CALL*** relocation causes the linker to build a global offset table if it has not already done so.

[f] An **R_MIPS_GOT16** for a local symbol must be followed immediately by an **R_MIPS_LO16**. Their combined AHL addend is used with the symbol value to calculate a relocated address. A GOT entry is constructed for the high-order 16 bits (using an existing one if possible), and its GOT offset becomes the value of the **R_MIPS_GOT16** relocation operator. The low-order 16 bits of the address becomes the value of the **R_MIPS_LO16** relocation operator.

[g] These operators relocate a 64-bit doubleword; all others relocate a 32-bit word.

[h] For these relocations, ld generates a *page pointer* in the GOT, i.e. an address within 32KB of (S+A). They are placed at small offsets from **gp** (i.e. within 32KB). **R_MIPS_GOT_PAGE** produces the GOT offset of the page pointer, and **R_MIPS_GOT_OFST** produces the offset of (S+A) from the page pointer.

[i] References to that location elsewhere are unchanged (i.e. they reference the new instruction) for the A form, or refer to the moved instruction for the B form. Relocations which follow an insertion relocation in the same record, or in consecutive records with a zero offset, reference the inserted instruction. Relocation records which follow with the same non-zero offset refer to the original operation at that address. This requires careful ordering or dummy intervening relocations if multiple relocations including insertions are to be applied to the first location in a section.

[j] References to the deleted address elsewhere are unchanged (i.e. they become references to the following object, which moves to the deleted address).

[k] The %higher(x) function is [(((long long) x + 0x80008000LL) >> 32) & 0xffff]. See the first example in the next section for the rationale for this definition and the %highest definition.

[l] The %highest(x) function is [(((long long) x + 0x800080008000LL) >> 48) & 0xffff].

Table 32 Relocation Types

Name	Value	Field	Symbol	Calculation

[m] The difference between the **R_MIPS_CALL*** operators and the corresponding **R_MIPS_GOT*** operators is that the former allow initial resolution by rld to a lazy evaluation stub, whereas the latter must be resolved to the ultimate address at initialization.

[n] An **R_MIPS_SCN_DISP** relocation is intended for address reset records in an Event Location section (see Section 2.10). Prior to relocation in the linker, the low-order 31 bits contains an offset from the beginning of the section referenced by the Event Location section's **sh_link** field. As the referenced sections and their associated Event Location sections are concatenated, these offsets must be updated to be relative to the merged section start address. Thus, "scn_addr" in the expression is the starting address of the section where the symbol is defined.

[o] **R_MIPS_ADD_IMMEDIATE** was used in Delta C++ to add a constant to a Delta symbol in an **ADDI** instruction. It is obsolete — **R_MIPS_16** should be used instead.

[p] An **R_MIPS_JALR** relocation is intended for optimization of jumps to protected symbols, i.e. symbols which may not be preempted. The word to be relocated is a jump (typically a JALR) to the indicated symbol. If it is not a preemptible symbol (and therefore defined in the current executable/DSO) the relocation is a request to the linker to convert it to a direct branch (typically a JAL in the main executable, or a BGEZAL in DSOs if the target symbol is close enough). The linker must check that the symbol is not preemptible before performing the relocation, but no action is required for correctness -- this is strictly an optimization hint.

[q] An **R_MIPS_RELGOT** relocation is the same as an **R_MIPS_REL** relocation, but relocates an entry in a GOT section and must be used for multigot GOTs (and only there).

An ABI-compliant object file must observe the ordering constraints from the footnotes to Table 32 in **SHT_REL** sections. Except as noted there, there are no constraints on the order of relocation operations in a section; using **SHT_RELA** sections eliminates even those constraints. However, producers should note that maintaining virtual address order (i.e. of the data objects to be relocated) will generally result in the best performance.

The new **R_MIPS_SHIFTn** operators are intended to support generation of shift instructions for the extraction of bitfields in C++ when the location of the bitfield in the object is unknown at compile time and determined only when an external class definition is linked in.

As currently defined, we implicitly assume that all GOT entries constructed in a 64-bit ELF object file are 64-bits, since we provide no relocation types to produce 32-bit GOT entries. It may prove desirable to provide both capabilities to allow smaller GOTs for programs residing entirely in the low 2GB of memory.

ELF-32

ELF-32: [To be supplied]

2.9.3 Examples

Following are a number of examples of relocation situations, with the relocations required in the 32-bit and 64-bit implementations.

Loading a 64-bit address: We have a symbol value (i.e. an address) which we wish to load into a register without going to memory. There are at least two possible sequences which may be desirable:

```
A:  lui    rx,%highest(sym)       # load highest "halfword"
    daddiu rx,rx,%higher(sym)     # merge next "halfword"
    dsll   rx,rx,16               # shift by one halfword
    daddiu rx,rx,%hi(sym)         # merge next "halfword"
    dsll   rx,rx,16               # shift into final position
    daddiu rx,rx,%lo(sym)         # merge lowest "halfword"

B:  lui    rx,%highest(sym)       # load highest "halfword"
    daddiu rx,rx,%higher(sym)     # merge next "halfword"
    dsll   rx,rx,32               # shift into high word
    lui    ry,%hi(sym)            # load high "halfword"
    daddiu ry,ry,%lo(sym)         # merge low "halfword"
    dadd   rx,rx,ry               # merge high + low words
```

These two sequences are equivalent. The first uses only one register, but is completely sequential (6 cycles minimum). The second uses a second register, but allows for a parallel schedule with a 4 cycle critical path on a superscalar processor.

Note that the somewhat odd definition of the %hi, %higher, and %highest relocation operations is necessary to make these sequences work, given that immediate add operations always use sign-extended immediates and that the lui operation sign-extends its result.

got_disp: We have a symbol (i.e. an address) which ld is to insert (as a pointer datum) into the GOT. We want to use the offset of that pointer from gp, e.g. as the offset (normally 16-bit) in a load instruction:

```
A:  ld  rx, %got_disp(sym) (gp)   # load address of sym
B:  ld  ry, 0(rx)                 # load object at sym
```

The relocation used in the 32-bit ABI, valid only for external symbols, is:

ext **sym**: **A: R_MIPS_GOT16(sym)**

We support new relocations in either 32-bit or 64-bit objects for arbitrary symbols:

any **sym**: **A: R_MIPS_GOT_DISP**(**sym**)

The 32-bit ABI provides no way of doing this for a local **sym**. See **got_page** and **got_ofst** below. Observe that although **R_MIPS_GOT_DISP** normally produces a validated 16-bit field, composing it with other operators allows its use to produce an arbitrarily sized GOT displacement.

got_hi (got_lo): As for **got_disp**, we want to reference the GOT displacement of a symbol address placed in the GOT by ld. However, the displacement may be larger than 16 bits, and this operator references the high-order (low-order) 16 bits:

```
A:  lui     rx, %got_hi(sym)         # load high part of disp
B:  dadd    rx, rx, gp               # add gp
C:  ld      rx, %got_lo(sym), rx     # load GOT entry
```

The relocations used in either 32-bit or 64-bit objects are:

any **sym**: **A: R_MIPS_GOT_HI16**(**sym**)

C: R_MIPS_GOT_LO16(**sym**)

Observe that this relocation is not defined in the 32-bit ABI, which in general does not cope with GOTs larger than 64KB.

If the symbol involved is a subprogram name being used in a call, which may therefore be resolved by ld for lazy evaluation, then the plain HI/LO relocations should be composed with **R_MIPS_CALL**, i.e.:

any **sym**: **A: R_MIPS_CALL(sym)**

R_MIPS_HI16(null)

C: R_MIPS_CALL(sym)

R_MIPS_LO16(null)

got_page (got_ofst): In some cases, we use the GOT entry (i.e. the symbol address) in a context where a 16-bit displacement can be added at the time of use, e.g. data loads. In such cases, far fewer GOT entries may be required if we store one address per 64KB page instead of one per address referenced, and use an offset from the page pointer in the final reference. We assume that the page entries can always be referenced within a 16-bit offset from **gp**, yielding sequences such as:

```
A:  lui     rx, %got_page(sym), gp# load page pointer
B:  ld      rx, %got_ofst(sym), rx   # load datum
```

The 32-bit ABI supports such sequences only for local symbols, using the following relocations:

local **sym**: **A: R_MIPS_GOT16(sym+addend)**
B: R_MIPS_LO16(sym+addend)

We support new relocations in either 32-bit or 64-bit objects for arbitrary symbols:

any **sym**: **A: R_MIPS_GOT_PAGE(sym+addend)**
B: R_MIPS_GOT_OFST(sym+addend)

Using new relocation types allows use for external symbols, potentially expanding the GOT size savings. If the referenced external turns out to be preemptible, ld should resolve its page pointer (i.e. the GOT entry) to its actual address and its offset to zero, effectively treating it like a normal GOT entry for an external symbol.

Observe that **R_MIPS_GOT_DISP**, **R_MIPS_GOT_PAGE**, and **R_MIPS_GOT_OFST** between them cover the local and external symbol cases handled in the 32-bit ABI by **R_MIPS_GOT16**. Separating them allows extension of each case to arbitrary symbols, without attempting to redefine **R_MIPS_GOT16**. The latter should fall into disuse except where it is required for 32-bit ABI conformance.

gp_rel: In some contexts, we need the (runtime) difference between a symbol address and the **gp**. The first is PIC branch tables, which are stored as the desired branch target addresses minus **gp** — by adding **gp** at runtime, we avoid having ***rld*** relocate the addresses. Thus, we have:

```
A:  %gp_rel(label1)
    %gp_rel(label2)
    ...
```

The relocation used for this purpose in the 32-bit ABI is:

local **sym**: **A: R_MIPS_GPREL32(sym)**

We also use **R_MIPS_GPREL32** in 64-bit objects. It will still yield a 32-bit displacement from **gp** by itself. Either it or **R_MIPS_GPREL** yields a 64-bit offset when composed with **R_MIPS_64**:

any **sym**: **A: R_MIPS_GPREL(sym)**
A: R_MIPS_64(null)

The second common case involves loading the difference between the entry address of a subprogram **s** and the runtime **gp** for establishing **gp**:

```
A:  lui    rx, %hi(%neg(%gp_rel(s)))# load high part of diff
B:  daddiu rx, rx, %lo(%neg(%gp_rel(s)))# add low part
C:  dadd   gp, t9, rx              # add to entry address
```

The 32-bit ABI handles this with a special case based on a reserved symbol **__gp_disp**, and requires that A and B be adjacent instructions:

__gp_disp: **A: R_MIPS_HI16(__gp_disp, addend)**
B: R_MIPS_LO16(__gp_disp, addend)

In 64-bit objects, we prefer composition:

any **s**: **A: R_MIPS_GPREL(s)**
A: R_MIPS_SUB(null)
A: R_MIPS_HI16(null)
B: R_MIPS_GPREL(s)
B: R_MIPS_SUB(null)
B: R_MIPS_LO16(null)

2.9.4 Discarded Relocations

The initial ***elf.h*** file defined several relocations for dealing with GOTs larger than 64KB which we do not include, favoring other approaches as described above:

R_MIPS_REL64
This can be produced by composing **R_MIPS_REL** with **R_MIPS_64**.

R_MIPS_LIT_HI16, R_MIPS_LIT_LO16
These can be produced by composing **R_MIPS_LITERAL** with **R_MIPS_HI16** or **R_MIPS_LO16**.

R_MIPS_GPOFF_HI16, R_MIPS_GPOFF_LO16

As described above, these can be produced by composing **R_MIPS_GPREL** with **R_MIPS_HI16** or **R_MIPS_LO16**.

2.10 Event Location Section

Stack traceback, as well as various transformations of object files, including PIC transformations, performance monitoring (pixie), processor bug workarounds (r4kpp), etc., require knowledge of where specific transitions occur in the program text. This section is intended to be a compact summary of this information (in conjunction with the DWARF frame information section, **.debug_frame**, which encodes the transitions of the stack pointer, frame pointer, and other registers). There will normally be one event location section per text section. Its entries must be in increasing address order.

Its section attributes are:

name	**.MIPS.events***name*
sh_type	**SHT_MIPS_EVENTS**
sh_link	Section header index of the (text) section described.
sh_info	Section header index of associated interface section.
sh_flags	**SHF_ALLOC + SHF_MIPS_NOSTRIP**
requirements	must not be stripped

The structure of an event location section is a sequence of variable-length records, each consisting of a kind byte followed by zero or more operands. The section alignment is 1 byte, and its size is the size of the actual data. The possible event kinds, along with their operands, are given by Table 33 below; the meanings are discussed in more detail below the table. The column giving operand types lists them in order of appearance when there is more than one operand for a particular kind. Note that the event kinds may not overlap with the content kinds given in Section 2.12 below, except for **EK_NULL**.

Table 33 Event Kind Constants

Event Kind Name	Value	Operand Type	Comments
EK_NULL	* *0x00*	**none**	No valid information — may be used as a filler

SiliconGraphics
Computer Systems

Table 33 Event Kind Constants

Event Kind Name	Value	Operand Type	Comments
EK_ADDR_RESET	* 0x01	**Elf64_Word** **Elf64_Half**	Reset current location to the given offset from the section (segment) start
EK_INCR_LOC_EXT	* 0x02	**ULEB128**	Increment current location by operand times 4.
EK_ENTRY	* 0x03	**none**	Subprogram entrypoint
EK_IF_ENTRY	* 0x04	**Elf64_Word**	Subprogram entrypoint with given interface descriptor offset.
EK_EXIT	* 0x05	**none**	Subprogram exit
EK_PEND	* 0x06	**none**	Subprogram end (last instruction)
EK_SWITCH_32	* 0x07	**Elf64_Byte** **Elf64_Word** **ULEB128**	Switch jr with 32- / 64-bit table entries. Operands indicate whether GP-relative, table start address, and table size.
EK_SWITCH_64	* 0x08		
EK_DUMMY	* 0x09	**none**	unused
EK_BB_START	* 0x0a	**none**	Start of basic block
EK_INCR_LOC_UNALIGNED	* 0x0b	**ULEB128**	Increment current location by operand (not multiplied by four)
EK_GP_PROLOG_HI	* 0x0c	**Elf64_Half**	Establish high/low 16 bits of GP; opnd is %lo(_gp_disp) / %hi(_gp_disp)
EK_GP_PROLOG_LO	* 0x0d	**Elf64_Half**	
EK_GOT_PAGE	* 0x0e	**Elf64_Half**	Reference GOT page / offset; opnd is corresponding offset / page.
EK_GOT_OFST	* 0x0f	**Elf64_Half**	
EK_HI	* 0x10	**Elf64_Half**	Reference high/low 16 bits of 32-bit absolute address; opnd is corresponding %lo(address) / %hi(address)
EK_LO	* 0x11	**Elf64_Half**	
EK_64_HIGHEST	* 0x12	**Elf64_Xword**	Reference 16 bit pieces of 64-bit absolute address. The operand is the full address.
EK_64_HIGHER	* 0x13	**Elf64_Xword**	
EK_64_HIGH	* 0x14	**Elf64_Xword**	
EK_64_LOW	* 0x15	**Elf64_Xword**	
EK_GPREL	* 0x16	**none**	GP-relative reference
EK_DEF	* 0x17	**below**	Define a new event kind operand profile

Table 33 Event Kind Constants

Event Kind Name	Value	Operand Type	Comments
EK_FCALL_LOCAL	* 0x18	**Elf64_Word**	Call to local routine with given offset in current section
EK_FCALL_EXTERN	* 0x19	**Elf64_Half**	Call external routine at given GP index
EK_FCALL_EXTERN_BIG	* 0x1a	**Elf64_Half** **Elf64_Half**	Call external routine at given GP index, given by %hi/%lo pair
EK_FCALL_MULT	* 0x1b	**ULEB128**	Call to any of several routines, none or all known
EK_FCALL_MULT_PARTIAL	* 0x1c	**ULEB128**	Call to any of several routines, some known
EK_LTR_FCALL	0x1d	**Elf64_Word**	Instruction is call to *lazy_text_resolve*; argument is .dynsym index of callee
EK_PCREL_GOT0	0x1e	**Elf64_Half**	Instruction is loading high half of PC-relative displacement to GOT 0, entry 0; argument is instruction count to instruction providing low half
EK_MEM_COPY_LOAD	0x1f	**none**	Load for purposes of copying data
EK_MEM_COPY_STORE	0x20	**LEB128**	Store for purposes of copying data — operand is distance to paired load
EK_MEM_PARTIAL_LOAD	0x21	**Elf64_Byte**	Reference to subset of bytes loaded — 8 bits of operand give bytes used
EK_MEM_EAGER_LOAD	0x22	**none**	Load is speculative
EK_MEM_VALID_LOAD	0x23	**none**	Load of data known to be valid
EK_CK_UNUSED_NONE_0	* 0x50	**none**	Reserved for future use
EK_CK_UNUSED_NONE_1	* 0x51	**none**	Reserved for future use
EK_CK_UNUSED_NONE_2	* 0x52	**none**	Reserved for future use
EK_CK_UNUSED_NONE_3	* 0x53	**none**	Reserved for future use
EK_CK_UNUSED_NONE_4	* 0x54	**none**	Reserved for future use
EK_CK_UNUSED_16BIT_0	* 0x55	**Elf64_Half**	Reserved for future use
EK_CK_UNUSED_16BIT_1	* 0x56	**Elf64_Half**	Reserved for future use
EK_CK_UNUSED_16BIT_2	* 0x57	**Elf64_Half**	Reserved for future use

Table 33 Event Kind Constants

Event Kind Name	Value	Operand Type	Comments
EK_CK_UNUSED_16BIT_3	** 0x58*	**Elf64_Half**	Reserved for future use
EK_CK_UNUSED_16BIT_4	** 0x59*	**Elf64_Half**	Reserved for future use
EK_CK_UNUSED_32BIT_0	** 0x5a*	**Elf64_Word**	Reserved for future use
EK_CK_UNUSED_32BIT_1	** 0x5b*	**Elf64_Word**	Reserved for future use
EK_CK_UNUSED_32BIT_2	** 0x5c*	**Elf64_Word**	Reserved for future use
EK_CK_UNUSED_64BIT_0	** 0x5d*	**Elf64_Xword**	Reserved for future use
EK_CK_UNUSED_64BIT_1	** 0x5e*	**Elf64_Xword**	Reserved for future use
EK_CK_UNUSED_64BIT_2	** 0x5f*	**Elf64_Xword**	Reserved for future use
EK_CK_UNUSED_64BIT_3	** 0x60*	**Elf64_Xword**	Reserved for future use
EK_CK_UNUSED_64BIT_4	** 0x61*	**Elf64_Xword**	Reserved for future use
EK_CK_UNUSED_ULEB128_0	** 0x62*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_1	** 0x63*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_2	** 0x64*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_3	** 0x65*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_4	** 0x66*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_5	** 0x67*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_6	** 0x68*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_7	** 0x69*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_8	** 0x6a*	**ULEB128**	Reserved for future use
EK_CK_UNUSED_ULEB128_9	** 0x6b*	**ULEB128**	Reserved for future use
EK_VENDOR_LOW	*0x70*	**undefined**	Kinds in this range are reserved for vendor-specific use
EK_VENDOR_HIGH	*0x7f*		
EK_INCR_LOC	** 0x80*	**none ***	Increment current location by low 7 bits of kind value times 4.

Events generally describe the behavior of instructions at a *current location*. The current location is modified by one of four event kinds. An **EK_ADDR_RESET** event specifies a full address relative to the beginning of the section (segment), and must be the first event (which allows the linker to always append a new section without insertions). The second operand of an **EK_ADDR_RESET** event is the distance in bytes until the next **EK_ADDR_RESET** event (or the remaining size of the section for the last one), allowing faster scanning of merged events sections. An **EK_INCR_LOC** event increments the current location, with the increment embedded in the event kind field; **EK_INCR_LOC_EXT** does the same with an explicit operand — each multiplies its operand by four. An **EK_INCR_LOC_UNALIGNED** event is like an **EK_INCR_LOC_EXT**, except that the operand is not multiplied by four, so it can be used for non-full-word-aligned increments.

ISSUE !

A program unit normally contains one or more entrypoints, given by **EK_ENTRY** or **EK_IF_ENTRY** events, and exits (i.e. returns) given by **EK_EXIT** events pointing to their return branch instructions. Its last instruction is marked by an **EK_PEND** event. **Issue**: should there be an **EK_PBEGIN** event kind to mark the first instruction of a program unit in the case that it is not also an entrypoint?

Switch statements are noted by **EK_SWITCH_32** or **EK_SWITCH_64** events pointing to the associated jr instruction. The operands indicate (in order) whether the table contents are GP-relative (single-byte Boolean), the table address (a fullword, relative to the start address of the section containing the jr instruction), and the number of entries in the table (**ULEB128**).

New basic blocks are noted by **EK_BB_START** events. These should only be present for cases that tools like pixie cannot identify reliably.

GP establishment in the prolog is indicated by the **EK_GP_PROLOG_HI** and **EK_GP_PROLOG_LO** events, pointing to the instructions which actually reference those pieces of the GP. Similarly, all of the event kinds through EK_GPREL reflect relocations in the original relocatable object which might need to be redone for code transformations. We call these collectively *relocation events*. Note that most of them come in hi/lo pairs; the operands in these cases provide the parts of the target address which cannot be obtained from the event instruction.

Calls are labelled by several *call event* kinds: A call to the current section is marked by an **EK_FCALL_LOCAL** event. An external call is marked by an **EK_FCALL_EXTERN** event (16-bit GOT displacement) or an **EK_FCALL_EXTERN_BIG** event. These events encode the GOT addressing of the callee in their operands; note that the instructions which load the addresses normally also require relocation events. If the callee is not known, there are three possibilities. **EK_FCALL_MULT** followed by two or more of the specific events above indicates that the callee is one of the given possibilities. **EK_FCALL_MULT_PARTIAL** followed by zero or more of the specific events above indicates that the callee is either one of the given possibilities or some other unknown function. In both cases, the operand is the number of specific call events following.

Several *memory events* mark special cases of memory reference instructions for the use of memory debugging tools like Purify. Copies are indicated by an **EK_MEM_COPY_LOAD** at the load instruction and an **EK_MEM_COPY_STORE** at the store instruction, with the latter having an operand giving the offset (divided by four) for the associated load. (There may be multiple stores generated for one load. This allows the tool to copy the valid bits rather than checking the load for validity.

Bitfield use (either explicitly using C bitfields, or implicitly with variants of *var&mask*) leads to loads which do not really involve use of all of the bytes loaded. **EK_MEM_PARTIAL_LOAD** marks such a load with a mask of the bytes actually required to be valid. For loads intended only to allow insertion of a bitfield to be stored, the mask will be zero. For loads intended for bitfield extraction, the mask should indicate which bytes are occupied by the bitfield to be extracted.

When the compiler generates eager loads which are not actually used on all paths, they should be marked by **EK_MEM_EAGER_LOAD** events. When the compiler generates loads which are known to be of valid data (because it is known to have been stored earlier, or it is known to be initialized global data, e.g. the GOT), they may be marked by **EK_MEM_VALID_LOAD** to save checking. None of these memory events are predefined, so they must be preceded by **EK_DEF** events (see below).

ISSUE !

Issue: Should the memory events be segregated to a distinct events section so that tools which use the default events and those which use the memory events aren't impacted by the other data?

Event section processors (consuming tools) must be able to parse the events section even in the presence of new event kinds added in the future (ignoring the new events). Therefore, before using any events besides those with values marked by '*' in the table above (the *predefined* kinds), the new event kind(s) must be defined by an **EK_DEF** event. Its operands are, in order, the new event kind value, the number of operands it requires, and the type of each operand, each represented by a single unsigned byte (as given by Table 34 below). The kinds named

EK_CK_UNUSED_... are considered predefined kinds for this purpose, so that **EK_DEF** events will not be required to use them in the future. They may ultimately be used as either event or content kinds.

In order to keep consumer semantics simple, all definitions of a new event kind must have consistent profiles, and if it is not the predefined profile of an **EK_CK_UNUSED_**... event, the relevant **EK_DEF** event must precede it in each object file where it is used. This allows old consumers to simply use the most recent encountered definition, and new consumers with an understanding of the event kind to ignore the **EK_DEF** events for it. It requires prospective producers to coordinate their new event kind proposals with the MIPS compiler group.

The valid operand type specifiers for an **EK_DEF** event are given by Table 34 below.

Table 34 Event Operand Type Specifiers in **EK_DEF** Events

Operand Type	Value	Comments
EK_DEF_UCHAR	1	Unsigned character (byte)
EK_DEF_USHORT	2	Unsigned short (2 bytes)
EK_DEF_UINT	3	Unsigned int (4 bytes)
EK_DEF_ULONG	4	Unsigned long (8 bytes)
EK_DEF_ULEB128	5	Unsigned LEB128 (variable byte length)
EK_DEF_CHAR	6	Signed character (byte)
EK_DEF_SHORT	7	Signed short (2 bytes)
EK_DEF_INT	8	Signed int (4 bytes)
EK_DEF_LONG	9	Signed long (8 bytes)
EK_DEF_LEB128	10	Signed LEB128 (variable byte length)
EK_DEF_STRING	11	Null-terminated character string
EK_DEF_VAR	12	Variable -length operand, consisting of a two-byte length including the length, followed by the remaining bytes.

Linker Processing

Every effort is made to minimize the space used by the event section, including the use of LEB128 encoding where possible. However, an important constraint is that the linker must be able to construct the combined event section by simply relocating and concatenating the individual events sections from the component objects, without changing their sizes (and therefore without needing to parse them). This leads to the use of fixed-size operands in several contexts where smaller operands would usually be adequate in relocatable object files.

ISSUE !

Issue: 16 event kinds **EK_VENDOR_...** have been reserved for use by individual vendors. However, their use would make the resulting objects unusable by any other vendor which used the same kinds. Would it be better to assume that such usage would use a distinct section, and define a mechanism for associating this section with a specific vendor?

2.11 Interface Section

Anticipated checking for correctness by the linkers (***ld/rld***), as well as object file transformations, require information about subprogram interfaces, especially parameter profiles. This section is intended to provide this information. There should be one such section per object file (including executables and DSOs).

These descriptors shall be used to describe both actual subprograms and the parameter profiles of calls. In the latter case, various information will be missing, and once the call is verified to match the profile of a callee, references to its descriptor (e.g. from the events section) may usually be converted to references to the callee's descriptor and the call descriptor may be removed.

The information to be provided has variable length. Thus, the section's contents are organized as a sequence of variable-length descriptors, each with a fixed-length header possibly followed by variable-length data. The descriptors must be sorted by the symbol table index in field *symbol*. Each descriptor must be a multiple of 8 bytes in size, with null padding between descriptors as required.

The Interface Section's attributes are:

name	**.MIPS.interfaces**
sh_type	**SHT_MIPS_IFACE**
sh_link	The section header index of the associated symbol table

sh_info	0
sh_flags	**SHF_ALLOC + SHF_MIPS_NOSTRIP**
requirements	may not be stripped — may be required for future interface checking and transformation.

The structure of an interface descriptor fixed-length header is given by Table 35 below.

Table 35 Interface Descriptor Header

Field name	Type	Comments
symbol	**Elf64_Word**	Symbol table index of subprogram name, or 0 for an indirect call
attrs	**Elf64_Half**	Attributes: see Table 36
pcnt	**Elf64_Byte**	Parameter count [a] [b]
fpmask	**Elf64_Byte**	Mask of FP parameter registers [c]

Notes:

[a] The parameter count and parameter profile below describe the parameter profile as transformed (not as declared). They include implicit parameters inserted by the compiler, including function results converted to implicit result pointers passed as parameters.

[b] The parameter count includes the result if the **SA_FUNCTION** attribute is set, and the first descriptor is for the result. If the parameter count is 255, then the parameter list is preceded by a two-byte parameter count. See Figure 2.)

[c] This mask indicates which of the eight FP parameter registers are used to pass parameters instead of the corresponding integer registers. The function result is not considered if **SA_FUNCTION** is set. The lowest-order bits of the mask represent the first parameters, i.e. 0x01 is parameter #1 in $f12, 0x02 parameter #2 in $f13, etc.

The attributes of a subprogram are encoded in the *attrs* field as the following bits.

Table 36 Subprogram Attributes

Attribute mask	Value	Comments
SA_PROTOTYPED	`0x8000`	Does def or ref have prototype?
SA_VARARGS	`0x4000`	Is this a varargs subprogram?
SA_PIC	`0x2000`	Are memory references PIC?
SA_DSO_ENTRY	`0x1000`	Is subprogram valid DSO entry?
SA_ADDRESSED	`0x0800`	Is subprogram address taken?
SA_FUNCTION	`0x0400`	Does subprogram return a result? [a]
SA_NESTED	`0x0200`	Is subprogram nested?[b]
SA_IGNORE_ERROR	`0x0100`	Don't enforce consistency.[c]
SA_DEFINITION	`0x0080`	Is this a definition (not just call)?
SA_AT_FREE	`0x0040`	Register **at** is free at all branches.[d]
SA_FREE_REGS	`0x0020`	Free register mask follows.[e]
SA_PARAMETERS	`0x0010`	Parameter profile follows.[f]
SA_ALTINTERFACE	`0x0008`	Next descriptor is an alternate interface for this subprogram.[g]

Notes:

[a] This is specified as transformed by the compiler, not necessarily as declared, e.g. a subprogram treating a structure result by placing it in a buffer addressed as an implicit first parameter would not be encoded as a function.

[b] Because nested subprograms require a static link in addition to the usual declared parameters, if a definition has this attribute all calls should too, but the opposite condition is not necessary.

[c] Some subprograms may be known to be called inconsistently. This attribute indicates that any tools checking for inconsistencies should not reject the objects due to inconstencies for this subprogram.

[d] Register **at** is never live at a branch instruction in this subprogram. It may therefore be used by transformation programs like pixie if required to transform the branch or the shadow operation. The flag must be reset by such a tool if **at** is used.

Table 36 Subprogram Attributes

Attribute mask	Value	Comments

[e] A 32-bit free register mask precedes the parameter profile, specifying integer registers which are never used in this routine. A program transformation tool like pixie may use these registers, subject to the ABI assumptions about caller-saved registers. The mask must be updated by such a tool if registers are used. See Figure 2)

[f] Minimal checking may be achieved by including only the fixed-length part of the interface descriptor, and omitting the detailed parameter profile. Doing so changes the meaning of the *pcnt* field in the fixed-length header. See the description of required linker checking below.

[g] In some cases the compiler (or another tool) may choose to create an alternate interface to a subprogram with different attributes. This flag indicates that the next descriptor is for such an alternate interface (not yet used).

The variable-length part of an interface descriptor consists of a list of parameter descriptors following the fixed-length header, possibly preceded by a free register mask. It contains one descriptor for each parameter (formal for declarations, actual for calls), plus a leading descriptor for the result of functions. Ellipsis parameters are not present for varargs subprograms; the **SA_VARARGS** attribute indicates this case. After resolving a reference in a call, the compiler or linker should replace the caller's actual interface reference by a reference to the callee's formal interface if they match. (Observe that a varargs call will never match unless no variable parameters are passed. Therefore, the call descriptor should not be removed unless the callee is defined and non-preemptible, though it may be merged with others that have the same profile.)

The structure of the variable-length part of an interface descriptor has the form given in Figure 2 below.

Figure 2 Subprogram Parameter Profile

Size in bytes of profile if any other fields are present (2 bytes)
Parameter count if *pcnt* **== 255**, else **0** (2 bytes)
Free register mask if **SA_FREE_REGS** is set (4 bytes)
Result type descriptor if **SA_FUNCTION** is set (2+ bytes)
Parameter #1 type descriptor (2+ bytes)
Parameter #2 type descriptor (2+ bytes)
...
Parameter #*pcnt* type descriptor (2+ bytes)

Each type descriptor has the following form:

flags	qual. count	fundamental type	first qualifier	...
0 3	4 7	8 15	16 23	

Various parts of the parameter profile, as well as the whole profile, are optional. The following rules should clarify which parts may be omitted:

- The entire profile is omitted if **SA_PARAMETERS** and **SA_FREE_REGS** are not set, and *pcnt* is not 255. See the linker processing description of minimal checking below for the treatment in this case.
- If any profile component is present, the size field must be present.
- If *pcnt* is 255, the parameter count field must be present, equal to the number of parameters, including the result for functions. (Note that this is the number of parameter register equivalents if **SA_PARAMETERS** is not set -- see the linker processing description of minimal checking below.) If *pcnt* is smaller, but **SA_FREE_REGS** is set, the parameter count field is present (containing zero) for alignment.

- If **SA_FREE_REGS** is set, the free register mask must be present (as well as the parameter count field).
- If **SA_PARAMETERS** is set, the number of parameter type descriptors implied by *pcnt* or the parameter count field is present. The first one is the function result type if **SA_FUNCTION** is set. These descriptors are described in more detail below.
- Finally, if the full profile is not a multiple of eight bytes long, it is padded to a multiple of eight bytes with zeros (which are not included in the size).

Each parameter type descriptor begins with a halfword, the contents of which are given by Table 37 below. It contains one of the fundamental types from Table 2 above in the low-order byte plus several flags and a qualifier count in the high-order byte as described in Table 37. If the qualifier count is non-zero, it will be followed by that number of single-byte type qualifiers from Table 3 above. Those types with indeterminate length (e.g. **FT_struct**, **FT_union**) are always followed by the actual length (in bytes) preceding the qualifier list. If **PDM_SIZE** is not set, the length is an unsigned byte, where the maximum value (255) implies at least that length; if **PDM_SIZE** is set it is a 32-bit unsigned word (not necessarily aligned), where the maximum value (0xffffffff) implies at least that length.

Table 37 Parameter Descriptor Masks

Mask Name	Value	Comments
PDM_TYPE	**0x00ff**	Fundamental type of parameter
PDM_REFERENCE	**0x4000**	Reference parameter?
PDM_SIZE	**0x2000**	Type followed by explicit 32-bit byte size?
PDM_Qualifiers	**0x0f00**	Count of type qualifiers <<8

2.11.1 Linker Processing

Linker Processing

The interface descriptor section requires significant processing by the linker — its purpose is to provide link-time checking. This processing involves checking and compression. This facility has been designed to be used either for minimal checking with minimal space requirements, or for full checking.

Full checking: The compiler may generate a descriptor for every subprogram definition and potentially for every call. (Only one descriptor is required for multiple calls to the same subprogram, and if the callee is in the same compilation, the compiler can check parameters and omit the call descriptors entirely.) When the linker resolves a call to the callee's definition, it should check the call descriptor against the definition descriptor. The rules for compatibility are as follows:

1. If this is not a varargs routine (not **SA_VARARGS**), then the number of parameters should match. For each parameter the sizes should match, and whether it is integer or floating point should match.
2. For a 32-bit (old ABI) varargs routine (**SA_VARARGS**), the fixed parameters should match (except perhaps for the last one, which may be a ***varargs.h*** **va_alist** dummy parameter).
3. For a 64-bit varargs routine (**SA_VARARGS**), the fixed parameters should match (except perhaps for the last one, which may be a ***varargs.h*** **va_alist** dummy parameter), and there must be no floating point parameters in the variable part unless the call site had a prototype visible (**SA_PROTOTYPED**).

Compatibility failures should result in warnings rather than hard errors.

Once checking has been done, the linker may (and should) discard most of the descriptors. In general, call descriptors should be discarded if a definition descriptor is available. If there is no definition available, then the linker may verify that the calls are consistent and discard all but one.

Minimal checking: At a minimum, checking should identify cases where floating point parameters have been passed to the variable part of a varargs routine's parameter list; this will not work unless a prototype for the callee was available at the call site. This minimum check requires only the fixed-length part of the descriptors. It requires that the compiler emit descriptors for any varargs routine definitions, and that it emit descriptors for any calls to varargs routines or to routines without prototypes, where floating point actual parameters are passed in registers. These descriptors will not have the **SA_PARAMETERS** flag set, and their *pcnt* field should reflect the number of register equivalents (i.e. 64-bit pieces for aggregate parameters) used to pass parameters rather than the number of source-level parameters (plus 1 if **SA_FUNCTION** is set).

The linker then checks for floating point parameters passed to the variable part of a varargs parameter list using the *fpmask* information in the fixed-length header and the *pcnt* field. The first (high-order) *pcnt* bits of the *fpmask* field must match. The remaining bits of the caller's *fpmask* field must be clear. This check is limited to the parameters which may be passed in registers, since those passed in memory do not present varargs matching problems.

Note that, if the callee's descriptor has the **SA_PARAMETERS** flag set but a minimal test is being done, the actual parameter descriptors must be examined to determine the actual number of parameter registers, rather than the source-level parameter count which *pcnt* will give.

2.12 Section Content Classification

Various tools need to identify the location of code, addresses, and other data in a program for transformation purposes. This section kind provides such information about the contents of sections.

The Content Section's attributes are:

name	**.MIPS.content***name*
sh_type	**SHT_MIPS_CONTENT**
sh_link	The section header index of the section classified
sh_info	0
sh_flags	**SHF_MIPS_NOSTRIP**
requirements	may not be stripped — doing so would render some functionality unusable, such as pixie and cordX

A content section is required for each section relevant to program execution which contains data of other than its default class, as determined by the section's attribute flags:

SHF_EXECINSTR	executable code
SHF_MIPS_ADDR	address data of size implied by section element size
other	non-address data

Note that *address data* refers to storage initialized to relocatable addresses, not to user pointer data which is uninitialized or initialized to **NULL**.

A content section is organized like the events section described in Section 2.10, as a sequence of variable-length records, each consisting of a single-byte content kind followed by zero or more bytes of operand values. Each record applies to a current location, where the current location is controlled by event kinds as described in Section 2.10. Most content records refer to a range starting at the current location, with the length of the range (in bytes) given by a **ULEB128** operand.

The content kind is one of the choices from Table 38. They are described further below the table. Content kind values may not conflict with event kinds described in Section 2.10, except for **CK_NULL**. Note, however, that the event kinds for setting the current location and for defining new kinds are also used in a content section for the same purposes. In addition, the reserved predefined kinds **EK_CK_UNUSED_...** listed in Table 33 may ultimately be used as either event or content kinds.

Table 38 Content Kind Constants

Content Kind Name	Value	Operand Type	Comments
CK_NULL	** 0x00*	**none**	No valid information — may be used as a filler
CK_DEFAULT	** 0x30*	**Elf64_Byte**	Operand is default data type for section.
CK_ALIGN	** 0x31*	**ULEB128** **Elf64_Byte**	First operand is length of range; second is required alignment (exponent 0..63 of 2)
CK_INSTR	** 0x32*	**ULEB128**	Range contains instructions
CK_DATA	** 0x33*	**ULEB128**	Range contains non-address data
CK_SADDR_32	** 0x34*	**ULEB128**	Range contains simple 32-bit addresses
CK_GADDR_32	** 0x35*	**ULEB128**	Range contains GP-relative 32-bit addresses
CK_CADDR_32	** 0x36*	**ULEB128**	Range contains complex 32-bit addresses
CK_SADDR_64	** 0x37*	**ULEB128**	Range contains simple 64-bit addresses
CK_GADDR_64	** 0x38*	**ULEB128**	Range contains GP-relative 64-bit addresses
CK_CADDR_64	** 0x39*	**ULEB128**	Range contains complex 64-bit addresses
CK_NO_XFORM	** 0x3a*	**ULEB128**	No transformation allowed in range
CK_NO_REORDER	** 0x3b*	**ULEB128**	No reordering allowed in range
CK_GP_GROUP	*0x3c*	**ULEB128, UINT**	Text in range with length given by first operand references **GP** group given by second

Table 38 Content Kind Constants

Content Kind Name	Value	Operand Type	Comments
CK_STUBS	*0x3d*	**ULEB128**	Text in range is delayed resolution stub code

If the normal default kind for a section is not appropriate or optimal, a different default may be specified by a **CK_DEFAULT** record. Its operand is the default kind to be used, which is one of the content kind values from the table.

Most of the content kind descriptors describe the content of a range of locations. In all such cases, they apply to a range starting at the current location and including the range given by a **ULEB128** operand which specifies the length of the range in bytes.

If a range of data has specific alignment requirements which must be preserved by transforming tools like pixie, this can be specified by a **CK_ALIGN** record, which provides the length of the affected range and the required alignment. This is intended for cases like embedding double-precision floating point data in text sections, where alignment must be preserved even if a transformer adds an odd number of instructions.

Data content type different than the default for the section can be specified by records of type **CK_INSTR** .. **CK_CADDR_64**. For each, the single operand is the length (in bytes) of the range containing the specified kind of data. The distinction between simple, GP-relative, and complex address data concerns how fixups may be performed if the addressed virtual memory is moved. Once all relocation has been done, simple address data should contain virtual addresses; if the content of the memory addressed is moved elsewhere in the virtual address space, the virtual addresses may simply be changed to reflect that shift. GP-relative address data should contain virtual addresses relative to the global pointer. Complex address data is a function of addresses (e.g. the difference between two addresses). Modifying complex address data if the address space is rearranged will require reevaluating its relocation expression, and therefore requires that the relevant relocation information be retained and consulted.

Some code is sensitive to precise ordering (e.g. code which does LL/SC sequences for synchronization), and tolerates little or no transformation.

The **CK_NO_XFORM** record indicates a range which may not be transformed except to modify branch targets and other addresses — instruction sequences and registers may not be changed. The **CK_NO_REORDER** record indicates a range where reordering of instructions is not allowed, but register remapping is allowed and instructions may be inserted. Both kinds have an operand specifying the length of the affected range in bytes.

The **CK_GP_GROUP** record is used in object files with multiple GOTs to indicate which parts of the object are associated with which GOT. For a text section (i.e. when *sh_link* points to a text section), it indicates a range of text addresses within which references to the global pointer (**GP**) refer to a particular **GP** group. The first operand is a **ULEB128** length of the range (in bytes), and the second is a 32-bit unsigned index of the **GP** group to which it refers. For a data section (i.e. when *sh_link* points to a data section like **.rodata**), it indicates the range of data covered by the indicated **GP**. For a GOT section (i.e. when *sh_link* points to a **.got** section), the range length indicates the length of the GOT itself.

Content section processors (consuming tools) must be able to parse the ifcontent section even in the presence of new content kinds added in the future (ignoring the new content specifiers). Therefore, before using any content kinds besides those with values marked by '*' in the table above (the *predefined* kinds), the new content kind(s) must be defined by an **EK_DEF** entry. See Section 2.10 for a description of this event kind.

2.13 Comment Section

The comment section is reserved for revision control information (see [ABI32]). Its attributes are:

name	**.comment**
sh_type	**SHT_PROGBITS**
sh_link	**SHN_UNDEF**
sh_info	0
sh_flags	none
requirements	may be stripped

The contents of a **.comment** section will be a sequence of NULL-terminated strings with the format of each string being:

toolname:vendor:revision:object

where:

toolname is the name of a tool which was used during contruction of this object file. If it is empty, then the revision refers to the object name (normally a source file).

vendor is the vendor of the tool (or object) identified. SGI/MIPS will normally leave this field empty for components of the compiler toolset, e.g. the compilers, ***ar***, ***ld***, ***pixie***, etc., or will use MIPS otherwise.

revision is a revision number for the tool (or object) identified. Its format is unspecified, but for SGI/MIPS tools it will normally have the form *nnn.mmm*. In some cases, a timestamp might be appropriate.

object is a source file or object file name identifying the object to which the tool was applied. If empty, the containing object is implied.

If toolname, vendor, and revision are all empty, the last triple with a non-empty revision are implied for the given object. Observe that the format of these strings implies that *toolname*, *vendor*, and *revision* may not contain colons.

ISSUE !

Issue: The rules for linker treatment of the comment section must be defined. Some compression is likely to be desirable.

2.14 Note Section

The note section is provided for use by tools which need to mark an object file with information not forseen by this specification (see [ABI32]). Its attributes are:

name	**.note**
sh_type	**SHT_NOTE**
sh_link	**SHN_UNDEF**
sh_info	0
sh_flags	none (by default)
requirements	may normally be stripped**,** but doing so may render functionality unusable — producers may set **SHT_MIPS_NOSTRIP** attribute

The note section consists of a sequence of name/descriptor pairs, of variable length, with the format described in [ABI32], modified for 8-byte alignment, as described in Table 39 below. Whereas [ABI32] requires that the *desc* field be 4-byte aligned and padded to a multiple of 4 bytes, ELF-64 requires that it be 8-byte aligned and padded.

Table 39 Note Descriptor Format

Field	Type	Comments
namesz	**Elf64_Word**	Size in bytes of *name* field
descsz	**Elf64_Word**	Size in bytes of *desc* field
type	**Elf64_Word**	Producer-specific type indicator
name	**char[]**	Producer-defined null-terminated string
desc	**Elf64_Xword[]**	Producer-defined descriptor

Producers and consumers of .note section information should take care to avoid conflicts as follows:

- The name field contents should be chosen to minimize the likelihood of conflict with other users. SGI/MIPS producers will use names of the form "MIPS:producer".
- Producers must preserve existing descriptors from other (unknown) producers in the note section, in the order found. They may place their own descriptors wherever they wish in the output sequence, however. Therefore, descriptor contents must be designed to be independent of their position in the section.
- Consumers must be prepared to ignore descriptors from unknown producers.

2.15 Compact Relocation Section

This section, generated automatically by the linker (ld) in the ucode system, contains various relocation information required by tools like pixie for program transformation, in a compact form. Prior to Sherwood, this information was kept in the **.comment** section (in 32-bit programs). It is obsoleted in ELF-64 by the **.events** and **.debug_frame** sections.

The Compact Relocation Section's attributes are:

name	**.MIPS.compact_rel**
sh_type	**SHT_MIPS_COMPACT**
sh_link	**SHN_UNDEF**
sh_info	0
sh_flags	**none**
requirements	obsolete

Section 3 Program Linking and Loading

This section deals with aspects of the object file format specific to executable and DSO files (which we refer to collectively as *program* files), and with the processing required by the static linker ld(1) and the dynamic linker rld(1).

3.1 Linker (ld) Requirements

This section is obviously not an exhaustive list; it is intended to collect miscellaneous requirements which are not traditional and not obviously implied by the format description.

The intent of some of these requirements, along with the specification of most of the sections described above as having the **SHF_ALLOC** attribute by default, is to allow a program (or another process monitoring it at runtime, like a debugger) to access the information in its program file by simple references to its address space, rather than requiring that it explicitly read the program file.

3.1.1 Headers

The ELF header, program header table, and section header table will be allocated, i.e. they will be treated like sections with the **SHF_ALLOC** flag set. Although they are considered optional by [ABI32], section headers will be present in a MIPS ABI-compliant program file and may not be stripped.

3.1.2 Automatically Generated Names

For each section with the **SHF_MIPS_NAMES** attribute set, the linker will automatically generate hidden weak external symbols:

__elf_vaddr_*name*	equal to the virtual address of the section
__elf_size_*name*	equal to the size (in bytes) of the section

where "***name***" is the section name. If the section is not allocatable (**SHF_ALLOC**), their values will be zero. If one of these symbols is referenced, then the linker will set the corresponding section's **SHF_MIPS_NOSTRIP** flag.

The linker will also generate protected external symbols for the ELF header (with name ***__elf_header***) and the program header table (***__program_header_table***). These symbols may be referenced for any executable or DSO which is part of a process by using **dlsym (3X)**.

3.2 Program Header

The program header of an executable/DSO file consists of an array of descriptors, one per loadable segment plus a few extras. The structure (from [ABI64]) is as follows:

Table 40 Elf64_Phdr Structure

Field Name	Type	Description
p_type	**Elf64_Word**	Segment descriptor type — see Table 41
p_flags	**Elf64_Word**	Flags for segment — see Table 42
p_offset	**Elf64_Off**	File offset of segment
p_vaddr	**Elf64_Addr**	Virtual start address
p_paddr	**Elf64_Addr**	Physical start address
p_filesz	**Elf64_Xword**	Byte size in file (may be zero)
p_memsz	**Elf64_Xword**	Byte size in memory (may be zero)
p_align	**Elf64_Xword**	Required alignment — see [ABI32]

The segment types in the *p_type* field are given by the following table:

Table 41 Elf64_Phdr Segment Types (*p_type*)

Name	Value	Description
PT_NULL	`0`	Null descriptor — ignore
PT_LOAD	`1`	Loadable segment
PT_DYNAMIC	`2`	Dynamic segment — see Table 44
PT_INTERP	`3`	Interpreter pathname
PT_NOTE	`4`	Auxiliary information segment
PT_SHLIB	`5`	Reserved
PT_PHDR	`6`	Program header segment
PT_LOPROC	`0x70000000`	First processor-specific type
PT_HIPROC	`0x7fffffff`	Last Processor-specific type
PT_MIPS_REGINFO	`0x70000000`	Register information segment
PT_MIPS_OPTIONS	`0x70000001`	Options segment

The segment flag bits in the *p_flags* field are given by the following table:

Table 42 Elf64_Phdr Segment Flags (*p_flags*)

Name	Value	Description
PF_X	`0x1`	Executable
PF_W	`0x2`	Writable
PF_R	`0x4`	Readable
PF_MASKPROC	`0xf0000000`	Processor-specific flags
PF_MIPS_LOCAL	`0x10000000`	Thread-local data — see Table 9

3.2.1 Segment Contents

A MIPS executable or DSO typically have a segment layout similar to the following, although this specification should not be construed to require a particular layout:

Headers: ELF header
Program header
Section headers

Text: .reginfo
.dynamic
.liblist
.rel.dyn
.conflict
.dynstr
.dynsym
.hash
<debug information sections>
.rodata
.text

Data:	.sdata
	.litX
	.got
	.data
	.bss

The following are constraints on the memory layout of a MIPS executable or DSO file or memory image.

1. The **gp** value must be within 2GB of any executable code (mandatory). This guarantees that the **gp** may be established using a 32-bit offset from the entry point of any function in register t9.
2. Any sections with the **SHF_MIPS_GPREL** flag must be allocated entirely within 32KB of the **gp** value (mandatory). This will normally include any .sdata or .litX sections, and possibly .got sections.
3. The executable code for a single executable or DSO may never be larger than 256MB, and it may never be loaded across a 256MB boundary.

The linker (***ld***) should normally group sections into segments according to the following rules:

1. Sections with the same name and attributes should be grouped together.
2. Groups from (1) with the same attributes should then be grouped.
3. Groups from (2) may then be grouped if their attributes are consistent with inclusion in a common segment. The rules for such grouping may be system-specific, and must balance the benefits of precise segment attributes against improved performance from limiting the number of segments to be loaded at runtime.

3.3 Dynamic Linking

This section discusses the data structures and issues relevant to dynamic linking.

3.3.1 Dynamic Section

The Dynamic Section's attributes are:

name	**.dynamic**

sh_type	**SHT_DYNAMIC**
sh_link	The section header index of the associated string table
sh_info	0
attributes	**SHT_ALLOC + SHT_MIPS_NOSTRIP**
requirements	may not be stripped

The **.dynamic** section, which must be identified by a **PT_DYNAMIC** segment descriptor for any executable or DSO with DSO dependencies, consists of a table of pairs with the following structure:

Table 43 Dynamic Structure (*Elf64_Dyn*)

Field Name	Field Type	Description
d_tag	**Elf64_Xword**	Kind — see
d_un	**union of:**	
d_val	**Elf64_Xword**	Kind-dependent value
d_ptr	**Elf64_Addr**	Kind-dependent address

The possible tag values, which union element they require, and whether they are present in executables and/or DSOs, are given by the following table:

Table 44 Dynamic Array Tags (*d_tag*)

Tag Name	Value	d_un	Executable	Shared Object
DT_NULL	0	`ignored`	mandatory	mandatory
DT_NEEDED	1	`d_val`	optional	optional
DT_PLTRELSZ	2	`d_val`	optional	optional
DT_PLTGOT	3	`d_ptr`	mandatory [a]	mandatory [a]
DT_HASH	4	`d_ptr`	mandatory	mandatory
DT_STRTAB	5	`d_ptr`	mandatory	mandatory
DT_SYMTAB	6	`d_ptr`	mandatory	mandatory
DT_RELA	7	`d_ptr`	mandatory	optional

Table 44 Dynamic Array Tags (*d_tag*)

Tag Name	Value	d_un	Executable	Shared Object
DT_RELASZ	`8`	`d_val`	mandatory	optional
DT_RELAENT	`9`	`d_val`	mandatory	optional
DT_STRSZ	`10`	`d_val`	mandatory	mandatory
DT_SYMENT	`11`	`d_val`	mandatory	mandatory
DT_INIT	`12`	`d_ptr`	optional	optional
DT_FINI	`13`	`d_ptr`	optional	optional
DT_SONAME	`14`	`d_val`	ignored	optional
DT_RPATH	`15`	`d_val`	optional	optional [a]
DT_SYMBOLIC	`16`	`ignored`	mandatory	`ignored`
DT_REL	`17`	`d_ptr`	mandatory	optional
DT_RELSZ	`18`	`d_val`	mandatory	optional
DT_RELENT	`19`	`d_val`	mandatory	optional
DT_PLTREL	`20`	`d_val`	optional	optional
DT_DEBUG	`21`	`d_ptr`	optional	ignored
DT_TEXTREL	`22`	`ignored`	optional	optional
DT_JMPREL	`23`	`d_ptr`	optional	optional
DT_LOPROC	`0x70000000`	`unspecified`	`unspecified`	`unspecified`
DT_HIPROC	`0x7fffffff`	`unspecified`	`unspecified`	`unspecified`
DT_MIPS_RLD_VERSION	`0x70000001`	`d_val`	mandatory	mandatory
DT_MIPS_TIME_STAMP	`0x70000002`	`d_val`	optional	optional
DT_MIPS_ICHECKSUM	`0x70000003`	`d_val`	optional	optional
DT_MIPS_IVERSION	`0x70000004`	`d_val`	optional	optional
DT_MIPS_FLAGS	`0x70000005`	`d_val`	mandatory	mandatory
DT_MIPS_BASE_ADDRESS	`0x70000006`	`d_ptr`	mandatory	mandatory
DT_MIPS_MSYM	`0x70000007`	`d_ptr`	optional	optional
DT_MIPS_CONFLICT	`0x70000008`	`d_ptr`	optional	optional
DT_MIPS_LIBLIST	`0x70000009`	`d_ptr`	optional	optional
DT_MIPS_LOCAL_GOTNO	`0x7000000a`	`d_val`	mandatory	mandatory
DT_MIPS_CONFLICTNO	`0x7000000b`	`d_val`	optional	optional
DT_MIPS_LIBLISTNO	`0x70000010`	`d_val`	optional	optional

Table 44 Dynamic Array Tags (*d_tag*)

Tag Name	Value	d_un	Executable	Shared Object
DT_MIPS_SYMTABNO	0x70000011	d_val	mandatory	mandatory
DT_MIPS_UNREFEXTNO	0x70000012	d_val	optional	optional
DT_MIPS_GOTSYM	0x70000013	d_val	mandatory	mandatory
DT_MIPS_HIPAGENO	0x70000014	d_val	optional	optional
DT_MIPS_RLD_MAP	0x70000016	d_val	optional	optional
DT_MIPS_DELTA_CLASS	0x70000017	d_val	optional	optional
DT_MIPS_DELTA_CLASS_NO	0x70000018	d_val	optional	optional
DT_MIPS_DELTA_INSTANCE	0x70000019	d_val	optional	optional
DT_MIPS_DELTA_INSTANCE_NO	0x7000001a	d_val	optional	optional
DT_MIPS_DELTA_RELOC	0x7000001b	d_val	optional	optional
DT_MIPS_DELTA_RELOC_NO	0x7000001c	d_val	optional	optional
DT_MIPS_DELTA_SYM	0x7000001d	d_val	optional	optional
DT_MIPS_DELTA_SYM_NO	0x7000001e	d_val	optional	optional
DT_MIPS_DELTA_CLASSSYM	0x70000020	d_val	optional	optional
DT_MIPS_DELTA_CLASSSYM_NO	0x70000021	d_val	optional	optional
DT_MIPS_CXX_FLAGS	0x70000022	d_val	optional	optional
DT_MIPS_PIXIE_INIT	0x70000023	d_val	optional	optional
DT_MIPS_SYMBOL_LIB	0x70000024	d_val	optional	optional
DT_MIPS_LOCALPAGE_GOTIDX	0x70000025	d_val	optional	optional
DT_MIPS_LOCAL_GOTIDX	0x70000026	d_val	optional	optional
DT_MIPS_HIDDEN_GOTIDX	0x70000027	d_val	optional	optional
DT_MIPS_PROTECTED_GOTIDX	0x70000028	d_val	optional	optional
DT_MIPS_OPTIONS	0x70000029	d_ptr	mandatory	mandatory
DT_MIPS_INTERFACE	0x7000002a	d_ptr	optional	optional
DT_MIPS_DYNSTR_ALIGN	0x7000002b	d_val	optional	optional
DT_MIPS_INTERFACE_SIZE	0x7000002c	d_val	optional	optional
DT_MIPS_RLD_TEXT_RESOLVE_ADDR	0x7000002d	d_ptr	optional	optional
DT_MIPS_PERF_SUFFIX	0x7000002e	d_val	optional	optional
DT_MIPS_COMPACT_SIZE	0x7000002f	d_val	optional	optional
DT_MIPS_GP_VALUE	0x70000030	d_ptr	optional	optional

Table 44 Dynamic Array Tags (*d_tag*)

Tag Name	Value	d_un	Executable	Shared Object
DT_MIPS_AUX_DYNAMIC	`0x70000031`	`d_ptr`	optional	optional
[a] These requirements are different in [ABI32M] than in [ABI32].				

Some of the specific requirements for these tags include:

DT_NULL This tag must terminate the list of dynamic section tags.

DT_NEEDED This is a string table offset of a required library's name. There must be such an entry for each required DSO. See [ABI32].

DT_PLTGOT This member has the address of the **.got** section. It is mandatory for MIPS executables and DSOs. (A completely non-shared executable with no DSO dependencies might have no GOT, but ABI compliance requires use of the libc DSO.)

DT_HASH This member gives the symbol hash table address.

DT_STRTAB This member gives the string table address (the **.dynstr** section). The string table contains symbol names, library names, and other strings required in the executable/DSO.

DT_STRSZ This member gives the byte size of the **DT_STRTAB** table.

DT_SYMTAB This member gives the symbol table address (the **.dynsym** section). All entries in a 64-bit file are **Elf64_Sym** type (see Table 11)

DT_SYMENT This member gives the byte size of a **DT_SYMTAB** entry.

DT_MIPS_SYMTABNO

This member contains the number of entries in the **.dynsym** section.

DT_RELA This member gives the address of a relocation table with entry type **Elf64_Rela** (see Table 29). Elf-64 executables will normally use this type of relocation because it is required for some of the relocation types (e.g. **R_MIPS_HI16**), but an ABI-compliant system must cope with **Elf64_Rel** tables as well.

DT_RELASZ This member gives the byte size of the **DT_RELA** table.

DT_RELAENT This member gives the byte size of a **DT_RELA** entry.

DT_SONAME This member gives the string table offset of the containing DSO's name.

DT_RPATH This member gives the string table offset of a shared library search path. If it is present in a referenced DSO at static link time, it is included in the final executable's **DT_RPATH**.

DT_SYMBOLIC This member, if present, causes references within the containing DSO to be resolved locally if possible (i.e. it makes them all non-preemptible).

DT_TEXTREL This member, if absent, implies that runtime relocations will not change a non-writable segment.

DT_MIPS_RLD_VERSION

This member gives a version ID for the *Runtime Linker Interface*.

DT_MIPS_TIME_STAMP This member gives a timestamp.

DT_MIPS_ICHECKSUM

This member gives a checksum of all external strings (names?) and common sizes.

DT_MIPS_IVERSION

This member gives the string table index of a compatible version string.

DT_MIPS_FLAGS

This member contains MIPS-specific flags (see below).

DT_MIPS_BASE_ADDRESS

This member contains the base address assumed for the executable/DSO at static link time. It is used to adjust addresses (e.g. in the GOT) when a DSO is relocated at run time. It is the preferred address for quickstart purposes.

DT_MIPS_CONFLICT

This member contains the address of the **.conflict** section. It is mandatory if there is a **.conflict** section.

DT_MIPS_CONFLICTNO

This member contains the number of entries in the **.conflict** section. It is mandatory if **DT_MIPS_CONFLICT** is present.

DT_MIPS_LIBLIST

This member contains the address of the **.liblist** section.

DT_MIPS_LIBLISTNO

This member contains the number of entries in the **.liblist** section. It is required if **DT_MIPS_LIBLIST** is present.

DT_MIPS_LOCAL_GOTNO

This member contains the number of local GOT entries.

DT_MIPS_LOCALPAGE_GOTIDX

This member contains the index in the GOT of the first page table entry for a segment. There will be one per segment, in the same order as the segments in the segment table. They are mandatory if there are page table entries for any segment, and the value for a segment without any page table entries must be zero.

DT_MIPS_LOCAL_GOTIDX

This member contains the index in the GOT of the first entry for a local symbol. It is mandatory if there are local symbol entries.

DT_MIPS_HIDDEN_GOTIDX

This member contains the index in the GOT of the first entry for a hidden symbol. It is mandatory if there are hidden symbol entries.

DT_MIPS_PROTECTED_GOTIDX

This member contains the index in the GOT of the first entry for a protected symbol. It is mandatory if there are protected symbol entries.

DT_MIPS_UNREFEXTNO

This member contains the index into the dynamic symbol table of the first external symbol that is not referenced in the same object.

DT_MIPS_GOTSYM

This member contains the index into the dynamic symbol table of the first entry that corresponds to an external symbol with an entry in the GOT. See Section 3.5.

DT_MIPS_HIPAGENO

This member contains the number of page table entries in the GOT. It is used by profiling tools and is optional.

DT_MIPS_OPTIONS

This member contains the address of the Options section, containing various execution options. It is mandatory.

DT_MIPS_INTERFACE

This member contains the address of the **.MIPS.interface** section, describing subprogram interfaces. It is mandatory if there is such a section in the executable/DSO.

DT_MIPS_INTERFACE_SIZE

This member contains the size in bytes of the **.MIPS.interface** section. It is mandatory if there is such a section in the executable/DSO.

DT_MIPS_RLD_TEXT_RESOLVE_ADDR

If present, this member contains the link-time address of **_rld_text_resolve** to place in GOT entry 0. If absent (or if present and the value is not the same as the address in ***rld*** of the **_rld_text_resolve** function), then ***rld*** places the true address of **_rld_text_resolve** into GOT entry 0 at run-time.

DT_MIPS_SYMBOL_LIB

This optional member contains the address of the **.MIPS.symlib** section, describing a mapping from the **.dynsym** symbols to the DSOs where they are defined.

The following tags are not normally present in a MIPS object file:

DT_JMPREL This member gives the address of relocation entries associated solely with the procedure linkage table. If present, **DT_PLTREL** and **DT_PLTRELSZ** are also required.

DT_PLTRELSZ This member gives the total byte size of the relocation entries associated with the PLT. Mandatory if **DT_JMPREL** is present.

DT_REL This member gives the address of a relocation table with entry type **Elf64_Rel** (see Table 29, and **DT_RELA** above).

DT_RELSZ This member gives the byte size of the **DT_RELA** table.

DT_RELENT This member gives the byte size of a **DT_RELA** entry.

DT_PLTREL This member gives the kind of relocation (**DT_RELA** or **DT_REL**) in the procedure linkage table.

DT_MIPS_PERF_SUFFIX This member contains an index to the string table. ***Rld*** appends the specified string to the shared object name specified in any dlopen calls. (For example, pixie creates binaries and shared objects with suffix ".pixie". Although it changes the shared objects in the liblist of an object to include the correct suffix, it cannot change the pathnames passed by the program to ***dlopen***.)

DT_MIPS_PIXIE_INIT This member contains the address of an initialization routine created by ***pixie***. (**DT_INIT** cannot be used for this purpose is because ***pixie*** depends on a specific order for its initialization routines, which is different from the ABI-specified **DT_INIT** order.

DT_MIPS_COMPACT_SIZE This member contains the size of a ucode compact relocation header record, and is not present in **-n32** or **-64** ELF files..

ISSUE !

Issue: The above list of dynamic tag descriptions is not yet complete.

The MIPS-specific flags given by **DT_MIPS_FLAGS** are given by the following table:

Table 45 **DT_MIPS_FLAGS** Masks

Name	Value	Description
RHF_NONE	`0x00000000`	None
RHF_QUICKSTART	`0x00000001`	Use runtime loading shortcuts if possible (see Section 3.8)
RHF_NOTPOT	`0x00000002`	Hash size not a power of two
RHF_NO_LIBRARY_REPLACEMENT	`0x00000004`	Ignore **LD_LIBRARY_PATH**
RHF_NO_MOVE	`0x00000008`	DSO addresses may not be relocated by ***rld***
RHF_SGI_ONLY	`0x00000010`	Contains SGI-specific features
RHF_GUARANTEE_INIT	`0x00000020`	Guarantee that **.init** will finish executing before any non-**.init** code in the DSO is called
RHF_DELTA_C_PLUS_PLUS	`0x00000040`	Contains Delta C++ code
RHF_GUARANTEE_START_INIT	`0x00000080`	Guarantee that **.init** will begin executing before any non-**.init** code in the DSO is called
RHF_PIXIE	`0x00000100`	Generated by ***pixie***
RHF_DEFAULT_DELAY_LOAD	`0x00000200`	Delay-load DSO by default
RHF_REQUICKSTART	`0x00000400`	Object may be requickstarted
RHF_REQUICKSTARTED	`0x00000800`	Object has been requickstarted
RHF_CORD	`0x00001000`	Generated by ***cord***
RHF_NO_UNRES_UNDEF	`0x00002000`	Object contains no unresolved **undef** symbols
RHF_RLD_ORDER_SAFE	`0x00004000`	Symbol table is in a safe order

ISSUE !

Issue: Is **RHF_NOTPOT** obsolete?

3.4 Shared Object Dependencies

The *System V ABI* [ABI32] defines the default library search path to be ***/usr/lib***. The MIPS old 32-bit ABI [ABI32M], defines the default library search path to be ***/usr/lib:/lib:/lib/cmplrs/cc:/usr/lib/cmplrs/cc:/opt/lib***. The runtime loader (***rld***) overrides this default with the value of the environment variable **LD_LIBRARY_PATH** if set.

This 64-bit ABI defines the default library search path to be ***/usr/lib64:/lib64:/opt/lib64***. It is overridden by the environment variable **LD_LIBRARY64_PATH** if set, or if not by **LD_LIBRARY_PATH** if set.

This new 32-bit ABI defines the default library search path to be ***/usr/lib32:/lib32:/opt/lib32***. It is overridden by the environment variable **LD_LIBRARYN32_PATH** if set, or if not by **LD_LIBRARY_PATH** if set.

3.5 The Global Offset Table

The organization of the GOT generally follows that of [ABI32M]. It is essentially a table of addresses, 64 bits each. We summarize it here primarily for completeness.

The GOT itself is located by the **DT_PLTGOT** dynamic tag. It is logically two tables. The first (with **DT_MIPS_LOCAL_GOTNO** entries) consists of *local* GOT addresses, i.e. non-preemptible (*protected*) addresses defined within the executable/DSO. They are initialized to their quickstart values, and must be relocated if and only if the DSO is loaded at a different address than that given by its **DT_MIPS_BASE_ADDRESS** dynamic tag.

The second part of the GOT is the *global* GOT addresses, i.e. those which are undefined or preemptible. Each entry in this part has an associated symbol entry in the **.dynsym** section. Those symbols start at the symbol table index given by the **DT_MIPS_GOTSYM** dynamic tag, and are in the same order as the global GOT entries. If a symbol is defined in the DSO (but preemptible), the GOT entry will normally be initialized to a quickstart value. See Figures 5-9 and 5-10 of [ABI32M] for details of the treatment.

Sections in ***.o*** files containing addresses destined for the GOT must have the **SHF_MIPS_GPREL** attribute, and will normally have the **SHF_MIPS_MERGE** attribute (indicating that duplicates are to be removed). Code references to the GOT in ***.o*** files may need to cope with offsets from **gp** greater than 16 bits much more often than in 32-bit programs (because the GOT entries are twice the size). ABI-compliant objects should use 32-bit offsets if it is possible that they will be linked into programs with large GOTs.

Observe that it is acceptable to allocate non-GOT data at **gp**-relative addresses, although the current 32-bit system does not do so. Such data (e.g. the **.sdata**, **.sbss**, and **.litX** sections) should be allocated first in the global data area, since its reason for being allocated here is normally to achieve short-offset addressing.

3.6 Symbol Resolution

STB_LOCAL symbols are always resolved within the executable/DSO where they appear by the static linker. This is also the case with **STB_GLOBAL** or **STB_WEAK** symbols with export class **STO_INTERNAL**, **STO_HIDDEN**, or **STO_PROTECTED**. (See Section 2.5 for identification of these classes.) Other global or weak symbols, however, are *preemptible*, i.e. they may be resolved to a definition in a different object file (executable/DSO) by the dynamic linker. The rules for doing so are as follows:

1. Create a search order of the executable and DSOs which make up the running process. The executable is first. Next come the DSOs which it references, given by the **DT_NEEDED** tags in order of appearance. (If there is a **DT_LIBLIST** tag, the list in the **.liblist** section is used instead — it should therefore be consistent with the **DT_NEEDED** order.) Then the **DT_NEEDED** tags of these DSOs are searched, and so on recursively breadth-first until no new DSOs are identified.
2. Undefined non-COMMON symbol references are resolved by the first object file on the list which provides a strong symbol definition, if any, or if not by the first object file which provides a weak symbol definition.
3. Undefined COMMON symbol references are resolved by the first object file on the list which defines the symbol (i.e. provides initial values), if any (again giving priority to strong definitions), or if not by the first object file where it appears.

3.7 Relocation

As required in [ABI32M], there will typically be exactly one relocation section, named .rel.dyn, which will normally contain only **R_MIPS_REL32** relocations. However, we do not require this, and the dynamic linker must deal with any relocation sections given by **DT_REL** or **DT_RELA** tags, and with any legal relocation types.

Unlike the current MIPS systems, the GOT or other segments containing relocatable values should not be made writable (unless page sharing with writable data requires it) — if there is a **DT_TEXTREL** tag, the dynamic linker must be prepared to relocate objects in read-only pages. Also, the system must provide the process with a private copy of any pages which are relocated dynamically.

3.8 Quickstart and Process Initiation Optimizations

We use the same quick start-up mechanisms as [ABI32M]. The definition of the **.liblist** and **.conflict** sections is unchanged from the 32-bit version except that the **.conflict** section contains 64-bit addresses (type **Elf64_Addr**).

We impose the same ordering constraints as [ABI32M] as conditions for using Quickstart functionality:

- The GOT-mapped portion of the **.dynsym** section must be ordered by increasing values in the *st_value* field. This requires that the **.got** section have the same order, since it must correspond to the **.dynsym** section.
- The **.rel.dyn** section must have all local entries first, followed by the external entries. Within each of these subsections, the entries must be ordered by increasing symbol index.

3.8.1 MIPS Symbol Table Extension Section

A MIPS symbol table extension section is unchanged from [ABI32M]. Its purpose is to facilitate relocation which must occur in spite of quick-start. It has the following attributes:

name	**.msym**
sh_type	**SHT_MIPS_MSYM**
sh_link	Section header index of **.dynsym**, or 0 (see note below)
sh_info	Section header index of **.rel.dyn**, or 0 (see note below)
sh_flags	**SHF_ALLOC**
requirements	Must be present for quickstart

This section is an array of **Elf64_msym** elements, each corresponding to an entry in the **.dynsym** section. If it is present, the symbols in the **.dynsym** section must be ordered with all external symbols first, followed by all local symbols. Additionally, all symbols of the same name must have contiguous **.dynsym** entries. If this section does not exist, there are no ordering constraints on the **.dynsym** section (unless **RHF_RLD_ORDER_SAFE** is set in **DT_MIPS_FLAGS**, which implies that all UNDEF global entries will precede all non-UNDEF entries). The layout of **Elf64_msym** elements, shown in Table 46 below, is defined in header file ***/usr/include/msym.h***.

Note: SGI implementations today are inconsistent about what they put in the **sh_link** and **sh_info** fields. ***Rld*** does not depend on them, and tools should validate the indicated sections by section type before doing so. Future implementations should fill these fields as specified.

Table 46 Symbol Table Extension Structure (**Elf64_msym**)

Field name	Type	Comments
ms_hash_value	**Elf64_Word**	Precomputed hash value
ms_info	**Elf64_Word**	Additional information:
ms_info >> 8		First relocation section entry for the symbol.
ms_info & 0xff		Flags (see Table 47)

The *ms_hash_value* member contains the static link time precomputed hash value for the symbol, without the final step of reduction modulo the hash table size. If this section (**.msym**) is present, the hash table size must be a power of two, and the hash table index of a symbol is computed by simply AND'ing *ms_hash_value* with the hash table size minus 1.

The *ms_info* member contains two sub-members. The high-order 24 bits are the index of the first **.rel.dyn** entry for this symbol. The low-order byte contains the flags in Table 47 below.

Table 47 **ELF64_msym** Flag Masks

Mask Name	Value	Comments
MS_ALIAS	`0x01`	Symbol is an alias

The MS_ALIAS flag means that any relocations to the symbol inside this object must be resolved as if it were an undefined symbol, but references outside this object may resolve to this symbol.

3.8.2 Shared Object List Section

A **.liblist** section is unchanged from [ABI32M]. Its purpose is to facilitate preemption of symbols in quickstarted programs, which requires checking that the DSO version loaded is the same one used to quickstart. It is an array of structures providing identification information for the DSOs on which a quickstarted object depends.

The **.liblist** section has the following attributes:

name	**.liblist**
sh_type	**SHT_MIPS_LIBLIST**
sh_link	Section header index of **.dynstr**
sh_info	number of entries
sh_flags	**SHF_ALLOC**
requirements	Must be present for quickstart if conflicts exist

Each of the **.liblist** entries has the structure given by Table 48 below.

Table 48 Shared Object Information Structure (Elf64_lib)

Field name	Type	Comments
l_name	**Elf64_Word**	Shared object name (**.dynstr** index)
l_time_stamp	**Elf64_Word**	Timestamp
l_checksum	**Elf64_Word**	Sum of all externally visible symbols' string names and common sizes
l_version	**Elf64_Word**	Interface version (**.dynstr** index)
l_flags	**Elf64_Word**	Flags (see Table 47)

The *l_name* field specifies the name of a shared object. Its value is a string table index. The name may be a trailing component of a pathname specified in the **DT_RPATH** dynamic tag or the **LD_LIBRARY*_PATH** environment variable, or it may be a name containing '/' characters interpreted as relative to '.', or it may be a full pathname.

The *l_time_stamp* field is a 32-bit timestamp, which may be combined with the *l_checksum* value and the *l_version* string to form a unique id for this shared object.

The *l_version* field specifies the interface version, as a string in the **.dynstr** section. The version is a single string containing no colons (:). It is compared against a colon separated string of versions pointed to by a dynamic section entry of the shared object. Shared objects with matching names are considered incompatible if the interface version strings are deemed incompatible. An index value of zero means no version string is specified.

The *l_flags* field contains a set of 1-bit flags, defined in Table 49 below.

Table 49 Library List Flags, *l_flags*

Name	Value	Description
LL_NONE	0x00000000	None
LL_EXACT_MATCH	0x00000001	At runtime use a unique ID composed of *l_time_stamp*, *l_checksum*, and *l_version* fields to demand that the run-time dynamic shared library match exactly the shared library used at static link time. Set by the ***ld*** option *-exact_version*, or turned off by the ***ld*** option *-ignore_version*.
LL_IGNORE_INT_VER	0x00000002	At runtime, ignore any version incompatibilities between the dynamic shared library and the library used at static link time.
LL_REQUIRE_MINOR	0x00000004	The entry must match both major and minor revision numbers. Ignored for DSOs not marked **RHF_SGI_ONLY**. Turned on by the ***ld*** option *-require_minor*.
LL_EXPORTS	0x00000008	Whenever the containing DSO is exposed to the static linker, this DSO is too, as though it were explicitly listed on the linker command line.
LL_DELAY_LOAD	0x00000010	The named DSO is not loaded until some function in it is called via lazy function resolution, at which time ***rld*** does an *sgidladd()* of the DSO, and after which lazy function resolution proceeds as usual for functions in the DSO. Turned on by the ***ld*** options *-default_delay_load* or *-delay_load*.

Table 49 Library List Flags, *l_flags*

Name	Value	Description
LL_DELTA	`0x00000020`	Delta C++ library

At most one of **LL_EXACT_MATCH**, **LL_IGNORE_INT_VER**, or **LL_REQUIRE_MINOR** may be set for any particular **.liblist** entry. The result of combining them is undefined.

3.8.3 Conflict Section

A **.conflict** section is unchanged from [ABI32M]. Its purpose is to facilitate preemption of symbols in quickstarted programs. It is an array of indexes into the **.dynsym** section. Each identifies a symbol with attributes that conflict with a shared object on which it depends, either in type or size, such that this definition will preempt the shared object's definition. The dependent shared object is identified at static link time.

The **.conflict** section has the following attributes:

name	**.conflict**
sh_type	**SHT_MIPS_CONFLICT**
sh_link	Section header index of **.dynsym**, or 0 (see note below)
sh_info	0
sh_flags	**SHF_ALLOC**
requirements	Must be present for quickstart if conflicts exist

Note: SGI implementations today are inconsistent about what they put in the **sh_link** and **sh_info** fields. ***Rld*** does not depend on them, and tools should validate the indicated sections by section type before doing so. Future implementations should fill these fields as specified.

Each element of a .conflict section is an **Elf64_conflict** struct as given by Table 50 below.

Table 50 Conflict Structure (**Elf64_conflict**)

Field name	Type	Comments
c_index	**Elf64_Addr**	**.dynsym** index of conflicting symbol

3.8.4 Symbol Library Section

The **.MIPS.symlib** section is used to improve rld lookup performance. It is logically an extension of the external symbol (**.dynsym**) array and is an array in parallel to the **.dynsym** array. If this section exists for a DSO or executable it has one entry for each symbol. It is not required unless there are delay-loaded DSOs in a program.

The **.MIPS.symlib** section has the following attributes:

name	**.MIPS.symlib**
sh_type	**SHT_MIPS_SYMBOL_LIB**
sh_link	Section header index of **.dynstr**
sh_info	Section header index of **.liblist**
sh_flags	**SHF_ALLOC**
requirements	Optional

Each entry in this section is an unsigned index into the .**liblist** section, identifying which DSO satisfies external references to the corresponding **.dynsym** symbol. If the entry is zero, there is no information. Otherwise, the entry minus one is the **.liblist** index of a DSO satisfying the reference.

If the **.liblist** section contains no more than 254 entries, then each **.MIPS.symlib** entry is a single 8 bit unsigned value. If the the **.liblist** section contains more than 254 entries, then each **.MIPS.symlib** entry is a 16 bit unsigned value. (At present, no **.liblist** section with more than 64K entries is envisaged. If required, a decision will have to be made to either use 32-bit entries or some more space-efficient encoding.)

Section 4 Archive File Format

The archive file (i.e. ***ar***(4) format) may be used to collect arbitrary files; we are concerned here with the specific case where those files are ELF object files. This format is based on the System V ABI [ABI32]; in particular, the magic string and member header format are unchanged.

Unlike the COFF archive format, we do not generate an archive hash table, since the IRIX 6.0 linker (***ld***) does not use it.

The linker (***ld***) will work more efficiently when component object files (not their file headers) are 8-byte aligned. Generating tools (especially the compilers) are encouraged to arrange this by padding them, i.e. by increasing the length of the component files to a multiple of 8 bytes.

4.1 Basic File Format

An object file archive consists of the following sequence of components. In general, each must start on a 2-byte boundary, and is padded with a newline if necessary to make it even length. Its *ar_size* in its header, however, does not include the padding byte.

- The archive magic string, **ARMAG** ("!<arch>\n"), **SARMAG** (8) bytes long.

The remaining components are all preceded with a member header as specified by [ELF32].

- An optional archive symbol table. This table is discussed further below.
- An optional archive string table. Such a component has *ar_name* ="//" in its header, blank padded. The string table consists of a sequence of null-terminated names.
- Some number of "normal" member files. The *ar_name* field of such a component contains its filename, slash-terminated and blank-padded, if it fits. Otherwise it contains a slash followed by the decimal representation of the name's file offset in the archive string table.

4.2 Archive Symbol Table Components

We define below two symbol table component formats. These are the current 32-bit ABI format and an analogous 64-bit form for 64-bit ELF object files.

- 32-bit generic ABI symbol table (see [ABI32]).

  Such a component has *ar_name*="/" in its header, i.e. a null file name, blank padded. This symbol table consists of:

  - The number of symbols defined (a 32-bit count).
  - A sequence of 32-bit file offsets, one for each symbol, relative to the beginning of the archive file.
  - A sequence of null-terminated symbol names, one for each symbol.

  The sequences of file offsets and symbol names must correspond 1-1, and they must occur in the same order as their containing files in the archive, i.e. the file offsets must be in non-decreasing order. There may be multiple definitions of a single name (from different archived object files).

  This particular component must be 4-byte aligned, and hence must precede all normal files if present. It must be a multiple of 4 bytes in size; it should be null-padded if necessary, and *ar_size* in its header should include the padding. (The alignment and padding requirement is a modification to the [ABI32] format.)

- 64-bit generic ABI style symbol table.

  Such a component has *ar_name*="/SYM64/" in its header, blank padded. This name starts like a null name (slash) but then contains a string in the usual padding characters to identify the format.

  This symbol table consists of:

  - The number of symbols defined (a 64-bit count).
  - A sequence of 64-bit file offsets, one for each symbol, relative to the beginning of the archive file.
  - A sequence of null-terminated symbol names, one for each symbol.

  The sequences of file offsets and symbol names must correspond 1-1, and they must occur in the same order as their containing files in the archive, i.e. the file offsets must be in non-decreasing order. There may be multiple definitions of a single name.

  This form of the symbol table is only used for archives of 64-bit ELF object files; in such cases it replaces the generic ELF form.

This particular component must be 8-byte aligned, and hence must precede all normal files if present. It must be a multiple of 8 bytes in size; it should be null-padded if necessary, and *ar_size* in its header should include the padding. (The alignment and padding requirement is a modification to the [ABI32] format.)

4.3 Archive Hash Table Components

4.4 Discussion

We have made one significant extension to the [ABI32] definition:

- In order to deal with archive files potentially larger than 4GB in size, we have added a 64-bit analogue of the 32-bit symbol table component, extended in the obvious way, i.e. by changing the symbol count and file offsets to 64-bit values. It is distinguished from the 32-bit form only by a special string in padding area of its *ar_name* field. We intend to support this option only for archives of 64-bit ELF object files.

Initially, at least, MIPS tools will support the 32-bit forms of the symbol and hash tables for archives consisting solely of 32-bit ELF object files, and the 64-bit forms for archives consisting solely of 64-bit ELF object files. However, it may prove desirable to use the 32-bit forms even for 64-bit object files because they are much more compact than the 64-bit forms, and the 64-bit file offsets will be required only very rarely. Therefore, tools which must deal with either format anyway should avoid depending on this restriction if possible.

Bibliography

[ABI32] AT&T, ***SYSTEM V APPLICATION BINARY INTERFACE***, 1990, Unix Press (Prentice-Hall).

[ABI32M] AT&T, ***SYSTEM V APPLICATION BINARY INTERFACE MIPS Processor Supplement***, 1991, Unix Press (Prentice-Hall).

[ABI64] SPARC International, ***SYSTEM V APPLICATION BINARY INTERFACE Generic 64-Bit Extensions***, March7, 1992, Delta Document 1.20 (Draft).

[AsmPG] MIPS Computer Systems, Inc., ***Assembly Language Programmer's Guide***, May 1989 (Order Number 3201DOC).

[DWARF-1] Unix International, Programming Languages SIG, *DWARF Debugging Information Format*, Version 1.

[DWARF] Unix International, Programming Languages SIG, *DWARF Debugging Information Format*, Version 2, Revision 2.0.0 (July 27, 1993).

[IrisCG] Silicon Graphics Computer Systems, Inc., ***IRIS-4D Series Compiler Guide***, February 1992 (Order Number 007-0905-030).

[Symbols] http://sahara.mti/SGIABI/Symbols.html