Executable and Linking Format 3

The type of object files created by the SPARC assembler version for SunOS 5.*x* are now *Executable and Linking Format* (ELF) files. These relocatable ELF files hold code and data suitable for linking with other object files to create an executable or a shared object file, and are the assembler normal output. The assembler can also write information to standard output (for example, under the `-S` option) and to standard error (for example, under the `-V` option). The SPARC assembler creates a default output file when standard input or multiple files are used.

This chapter is organized into the following sections:

ELF Header	*page 12*
Sections	*page 14*
Locations	*page 21*
Relocation Tables	*page 22*
Symbol Tables	*page 22*
Addresses	*page 22*
String Tables	*page 24*
Assembler Directives	*page 24*

The ELF object file format consists of:

- Header
- Sections

3

- Locations
- Addresses
- Relocation tables
- Symbol tables
- String tables

For more information, see Chapter 4, "Object Files," in the *System V Application Binary Interface (SPARC™ Processor Supplement)* manual.

ELF Header

The *ELF header* is always located at the beginning of the ELF file. It describes the ELF file organization and contains the actual sizes of the object file control structures. The initial bytes of an ELF header specify how the file is to be interpreted.

The ELF header contains the following information:

`ehsize`
ELF header size in bytes.

`entry`
Virtual address at which the process is to start. A value of 0 indicates no associated entry point.

`flag`
Processor-specific flags associated with the file.

`ident`
Marks the file as an object file and provides machine-independent data to decode and interpret the file contents.

`machine`
Specifies the required architecture for an individual file. A value of 2 specifies SPARC.

`phentsize`

Size in bytes of entries in the program header table. All entries are the same size.

`phnum`

Number of entries in program header table. A value of 0 indicates the file has no program header table.

`phoff`

Program header table file offset in bytes. The value of 0 indicates no program header.

`shentsize`

Size in bytes of the section header. A section header is one entry in the section header table; all entries are the same size.

`shnum`

Number of entries in section header table. A value of 0 indicates the file has no section header table.

`shoff`

Section header table file offset in bytes. The value of 0 indicates no section header.

`shstrndx`

Section header table index of the entry associated with the section name string table. A value of `SHN_UNDEF` indicates the file does not have a section name string table.

`type`

Identifies the object file type. Table 3-1 describes the reserved object file types.

`version`

Identifies the object file version.

3

Table 3-1 shows reserved object file types:

Table 3-1 Reserved Object File Types

Type	Value	Description
`none`	`0`	No file type
`rel`	`1`	Relocatable file
`exec`	`2`	Executable file
`dyn`	`3`	Shared object file
`core`	`4`	Core file
`loproc`	`0xff00`	Processor-specific
`hiproc`	`0xffff`	Processor-specific

Sections

A section is the smallest unit of an object that can be relocated. The following sections are commonly present in an ELF file:

- Section header
- Executable text
- Read-only data
- Read-write data
- Read-write uninitialized data (*section header* only)

Sections do not need to be specified in any particular order. The *current section* is the section to which code is generated.

These sections contain all other information in an object file and satisfy several conditions.

1. Every section must have one section header describing the section. However, a section header does not need to be followed by a section.
2. Each section occupies one contiguous sequence of bytes within a file. The section may be empty (that is, of zero-length).

3. A byte in a file can reside in only one section. Sections in a file cannot overlap.

4. An object file may have inactive space. The contents of the data in the inactive space are unspecified.

Sections can be added for multiple text or data segments, shared data, user-defined sections, or information in the object file for debugging.

Note – Not all of the sections need to be present.

Section Header

The *section header* allows you to locate all of the file sections. An entry in a section header table contains information characterizing the data in a section.

The section header contains the following information:

`addr`

Address at which the first byte resides if the section appears in the memory image of a process; the default value is 0.

`addralign`

Aligns the address if a section has an address alignment constraint; for example, if a section contains a double-word, the entire section must be ensured double-word alignment. Only 0 and positive integral powers of 2 are currently allowed. A value of 0 or 1 indicates no address alignment constraints.

`entsize`

Size in bytes for entries in fixed-size tables such as the symbol table.

`flags`

One-bit descriptions of section attributes. Table 3-2 describes the section attribute flags.

Table 3-2 Section Attribute Flags

Flag	Default Value	Description
`SHF_WRITE`	`0x1`	Contains data that is writable during process execution.
`SHF_ALLOC`	`0x2`	Occupies memory during process execution. This attribute is *off* if a control section does not reside in the memory image of the object file.
`SHF_EXECINSTR`	`0x4`	Contains executable machine instructions.
`SHF_MASKPROC`	`0xf0000000`	Reserved for processor-specific semantics.

`info`

Extra information. The interpretation of this information depends on the section type, as described in Table 3-3.

`link`

Section header table index link. The interpretation of this information depends on the section type, as described in Table 3-3.

`name`

Specifies the section name. An index into the section header string table section specifies the location of a null-terminated string.

`offset`

Specifies the byte offset from the beginning of the file to the first byte in the section.

Note – If the section type is `SHT_NOBITS`, *offset* specifies the conceptual placement of the file.

`size`

Specifies the size of the section in bytes.

Note – If the section type is `SHT_NOBITS`, *size* may be non-zero; however, the section still occupies no space in the file.

`type`

Categorizes the section contents and semantics. Table 3-3 describes the section types.

Table 3-3 Section Types

Name	Value	Description	Interpretation by	
			info	**link**
`null`	`0`	Marks section header as inactive.		
`progbits`	`1`	Contains information defined explicitly by the program.		
`symtab`	`2`	Contains a symbol table for link editing. This table may also be used for dynamic linking; however, it may contain many unnecessary symbols. Note: *Only one section of this type is allowed in a file*	One greater than the symbol table index of the last local symbol.	The section header index of the associated string table.
`strtab`	`3`	Contains a string table. A file may have multiple string table sections.		
`rela`	`4`	Contains relocation entries with explicit addends. A file may have multiple relocation sections.	The section header index of the section to which the relocation applies.	The section header index of the associated symbol table.
`hash`	`5`	Contains a symbol rehash table. Note: *Only one section of this type is allowed in a file*	0	The section header index of the symbol table to which the hash table applies.
`dynamic`	`6`	Contains dynamic linking information. Note: *Only one section of this type is allowed in a file*	0	The section header index of the string table used by entries in the section.

Table 3-3 Section Types *(Continued)*

Name	Value	Description	Interpretation by	
			info	link
note	7	Contains information that marks the file.		
nobits	8	Contains information defined explicitly by the program; however, a section of this type does not occupy any space in the file.		
rel	9	Contains relocation entries without explicit addends. A file may have multiple relocation sections.	The section header index of the section to which the relocation applies.	The section header index of the associated symbol table.
shlib	10	Reserved.		
dynsym	11	Contains a symbol table with a minimal set of symbols for dynamic linking. Note: *Only one section of this type is allowed in a file*	One greater than the symbol table index of the last local symbol.	The section header index of the associated string table.
loproc hiproc	0x70000000 0x7fffffff	Lower and upper bound of range reserved for processor-specific semantics.		
louser hiuser	0x80000000 0xffffffff	Lower and upper bound of range reserved for application programs. Note: *Section types in this range may be used by an application without conflicting with system-defined section types.*		

Note – Some section header table indexes are reserved and the object file will not contain sections for these special indexes.

Predefined User Sections

A section that can be manipulated by the section control directives is known as a *user section*. You can use the section control directives to change the user section in which code or data is generated. Table 3-4 lists the predefined user sections that can be named in the section control directives.

Table 3-4 Predefined User Sections

Section Name	Description
`.bss`	Section contains uninitialized read-write data.
`.comment`	Comment section.
`.data & .data1`	Section contains initialized read-write data.
`.debug`	Section contains debugging information.
`.fini`	Section contains runtime finalization instructions.
`.init`	Section contains runtime initialization instructions.
`.rodata & .rodata1`	Section contains read-only data.
`.text`	Section contains executable text.
`.line`	Section contains line # info for symbolic debugging.
`.note`	Section contains note information.

Creating an `.init` Section in an Object File

The `.init` sections contain codes that are to be executed before the the main program is executed. To create an `.init` section in an object file, use the assembler pseudo-ops shown in Code Example 3-1.

```
.section ".init"
.align   4
<instructions>
```

Code Example 3-1 Creating an `.init` Section

At link time, the `.init` sections in a sequence of `.o` files are concatenated into an `.init` section in the linker output file. The code in the `.init` section are executed before the main program is executed.

Note – The codes are executed inside a stack frame of 96 bytes. Do not reference or store to locations that are greater than `%sp+96` in the `.init` section.

Creating a `.fini` *Section in an Object File*

`.fini` sections contain codes that are to be executed after the the main program is executed. To create an `.fini` section in an object file, use the assembler pseudo-ops shown in Code Example 3-2.

```
.section ".fini"
.align   4
<instructions>
```

Code Example 3-2 Creating an `.fini` Section

At link time, the `.fini` sections in a sequence of `.o` files are concatenated into a `.fini` section in the linker output file. The codes in the `.fini` section are executed after the main program is executed.

Note – The codes are executed inside a stack frame of 96 bytes. Do not reference or store to locations that are greater than `%sp+96` in the `.fini` section.

Predefined Non-User Sections

Table 3-5 lists sections that are predefined but cannot be named in the section control directives because they are not under user control.

Table 3-5 Predefined Non-User Sections

Section Name	Description
`".dynamic"`	Section contains dynamic linking information.

Table 3-5 Predefined Non-User Sections *(Continued)*

Section Name	Description
`.dynstr`	Section contains strings needed for dynamic linking.
`.dynsym`	Section contains the dynamic linking symbol table.
`.got`	Section contains the global offset table.
`.hash`	Section contains a symbol hash table.
`.interp`	Section contains the path name of a program interpreter.
`.plt`	Section contains the procedure linking table.
`.relname &` `.relaname`	Section containing relocation information. *name* is the section to which the relocations apply, that is, `".rel.text"`, `".rela.text"`.
`.shstrtab`	String table for the section header table names.
`.strtab`	Section contains the string table.
`.symtab`	Section contains a symbol table.

Locations

A *location* is a specific position within a section. Each location is identified by a section and a byte offset from the beginning of the section. The *current location* is the location within the current section where code is generated.

A *location counter* tracks the current offset within each section where code or data is being generated. When a section control directive (for example, `.section` pseudo-op) is processed, the location information from the location counter associated with the new section is assigned to and stored with the name and value of the current location.

The current location is updated at the end of processing each statement, but can be updated during processing of data-generating assembler directives (for example, the `.word` pseudo-op).

3

Note – Each section has one location counter; if more than one section is present, only one location can be current at any time.

Addresses

Locations represent *addresses in memory* if a section is allocatable; that is, its contents are to be placed in memory at program runtime. Symbolic references to these locations must be changed to addresses by the SPARC link editor.

Relocation Tables

The assembler produces a companion *relocation table* for each relocatable section. The table contains a list of relocations (that is, adjustments to data in the section) to be performed by the link editor.

Symbol Tables

A *symbol table* contains information to locate and relocate symbolic definitions and references. The SPARC assembler creates a symbol table section for the object file. It makes an entry in the symbol table for each symbol that is defined or referenced in the input file and is needed during linking. The symbol table is then used by the SPARC link editor during relocation. The section header contains the symbol table index for the first non-local symbol.

A symbol table contains the following information:

`name`

Index into the object file symbol string table. A value of zero indicates the symbol table entry has no name; otherwise, the value represents the string table index that gives the symbol name.

`value`

Value of the associated symbol. This value is dependent on the context; for example, it may be an address, or it may be an absolute value.

`size`

Size of symbol. A value of 0 indicates that the symbol has either no size or an unknown size.

`info`

Specifies the symbol type and binding attributes. Table 3-6 and Table 3-7 describes these values.

`other`

Undefined meaning. Current value is 0.

`shndx`

Contains the section header table index to another relevant section, if specified. As a section moves during relocation, references to the symbol will continue to point to the same location because the value of the symbol will change as well.

Table 3-6 Symbol Types

Value	Type	Description
0	`notype`	Type not specified.
1	`object`	*Symbol* is associated with a data object; for example, a variable or an array.
2	`func`	*Symbol* is associated with a function or other executable code. When another object file references a function from a shared object, the link editor automatically creates a procedure linkage table entry for the referenced symbol.
3	`section`	*Symbol* is associated with a section. These types of symbols are primarily used for relocation.
4	`file`	Gives the name of the source file associated with the object file.
13 15	`loproc` `hiproc`	Values reserved for processor-specific semantics.

Table 3-7 shows the symbol binding attributes.

Table 3-7 Symbol Bindings

Value	Binding	Description
0	`local`	*Symbol* is defined in the object file and not accessible in other files. Local symbols of the same name may exist in multiple files.
1	`global`	*Symbol* is either defined externally or defined in the object file and accessible in other files.
2	`weak`	*Symbol* is either defined externally or defined in the object file and accessible in other files; however, these definitions have a lower precedence than globally defined symbols.
13 15	`loproc` `hiproc`	Values reserved for processor-specific semantics.

String Tables

A *string table* is a section which contains null-terminated variable-length character sequences, or strings, in the object file; for example, symbol names and file names. The strings are referenced in the section header as indexes into the string table section.

- A string table index may refer to any byte in the section.
- Empty string table sections are permitted; however, the index referencing this section must contain zero.

A string may appear multiple times and may also be referenced multiple times. References to substrings may exist, and unreferenced strings are allowed.

Assembler Directives

Assembler directives, or pseudo-operations (pseudo-ops), are commands to the assembler that may or may not result in the generation of code. The different types of assembler directives are:

- Section Control Directives
- Symbol Attribute Directives
- Assignment Directives

- Data Generating Directives
- Optimizer Directives

See Appendix A, "Pseudo-Operations," for a complete description of the pseudo-ops supported by the SPARC assembler.

Section Control Directives

When a section is created, a section header is generated and entered in the ELF object file section header table. The *section control pseudo-ops* allow you to make entries in this table. Sections that can be manipulated with the section control directives are known as *user sections*. You can also use the section control directives to change the user section in which code or data is generated.

Note – The *symbol table*, *relocation table*, and *string table* sections are created implicitly. The section control pseudo-ops cannot be used to manipulate these sections.

The section control directives also create a section symbol which is associated with the location at the beginning of each created section. The section symbol has an offset value of zero.

Symbol Attribute Directives

The *symbol attribute* pseudo-ops declare the symbol type and size and whether it is local or global.

Assignment Directive

The *assignment* directive associates the value and type of expression with the symbol and creates a symbol table entry for the symbol. This directive constitutes a *definition* of the symbol and, therefore, must be the only definition of the symbol.

Data Generating Directives

The *data generating* directives are used for allocating storage and loading values.

3