ELF-64 Object File Format

Version 1.5 Draft 2
May 27, 1998

This document describes the current HP/Intel definition of the ELF-64 object file format. It is, for the most part, a simple extension of the ELF-32 format as defined originally by AT&T, although some fields have been rearranged to keep all fields naturally aligned without any internal padding in the structures.

Additional detail about the ELF-32 format may be obtained from any of the following sources:

- *Unix System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools*
- *System V Application Binary Interface, Revised Edition*
- *System V Interface Definition, Third Edition*
- *Tool Interface Standards: Portable Formats Specification, Version 1.0*

The processor-specific details of the ELF formats are covered in separate supplements. As much as possible, processor-specific definitions apply equally to ELF-32 and ELF-64.

Many implementations of ELF also include symbolic debug information in the DWARF format. We regard the choice of debug format as a separate issue, and do not include debug information in this specification.

1. Overview of an ELF file

An ELF object file consists of the following parts:

- File header, which must appear at the beginning of the file.
- Section table, required for relocatable files, and optional for loadable files.
- Program header table, required for loadable files, and optional for relocatable files. This table describes the loadable segments and other data structures required for loading a program or dynamically-linked library in preparation for execution.
- Contents of the sections or segments, including loadable data, relocations, and string and symbol tables.

Relocatable and loadable object files are illustrated in Figure 1. The contents of these parts are described in the following sections.

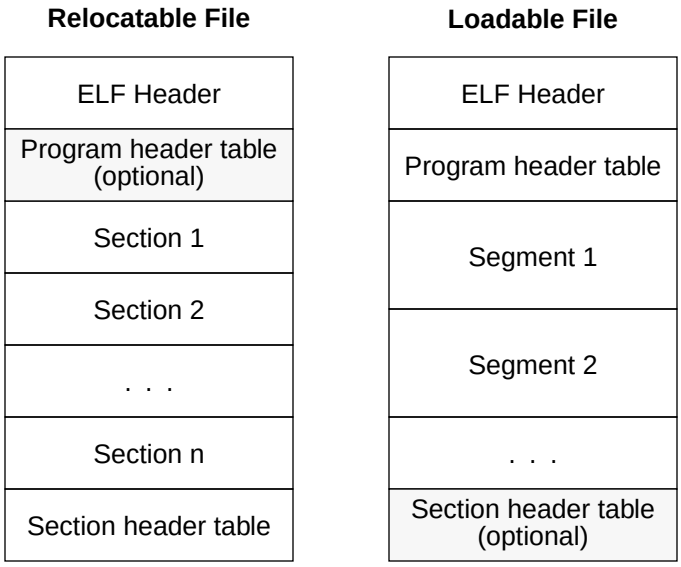

Figure 1. Structure of an ELF File

2. Data representation

The data structures described in this document are described in a machine-independent format, using symbolic data types shown in the Table 1. For 64-bit processors, these data types have the sizes and alignments shown.

Table 1. ELF-64 Data Types

Name	*Size*	*Alignment*	*Purpose*
Elf64_Addr	8	8	Unsigned program address
Elf64_Off	8	8	Unsigned file offset
Elf64_Half	2	2	Unsigned medium integer
Elf64_Word	4	4	Unsigned integer
Elf64_Sword	4	4	Signed integer
Elf64_Xword	8	8	Unsigned long integer
Elf64_Sxword	8	8	Signed long integer
unsigned char	1	1	Unsigned small integer

The data structures are arranged so that fields are aligned on their natural boundaries and the size of each structure is a multiple of the largest field in the structure without padding.

3. File header

The file header is located at the beginning of the file, and is used to locate the other parts of the file. The structure is shown in Figure 2.

```
typedef struct
{
        unsigned char   e_ident[16];    /* ELF identification */
        Elf64_Half      e_type;         /* Object file type */
        Elf64_Half      e_machine;      /* Machine type */
        Elf64_Word      e_version;      /* Object file version */
        Elf64_Addr      e_entry;        /* Entry point address */
        Elf64_Off       e_phoff;        /* Program header offset */
        Elf64_Off       e_shoff;        /* Section header offset */
        Elf64_Word      e_flags;        /* Processor-specific flags */
        Elf64_Half      e_ehsize;       /* ELF header size */
        Elf64_Half      e_phentsize;    /* Size of program header entry */
        Elf64_Half      e_phnum;        /* Number of program header entries */
        Elf64_Half      e_shentsize;    /* Size of section header entry */
        Elf64_Half      e_shnum;        /* Number of section header entries */
        Elf64_Half      e_shstrndx;     /* Section name string table index */
} Elf64_Ehdr;
```

Figure 2. ELF-64 Header

The fields in the ELF header have the following meanings:

- e_ident identify the file as an ELF object file, and provide information about the data representation of the object file structures. The bytes of this array that have defined meanings are detailed below. The remaining bytes are reserved for future use, and should be set to zero. Each byte of the array is indexed symbolically using the names in the Table 2.

Table 2. ELF Identification, e_ident

Name	*Value*	*Purpose*
EI_MAG0	0	File identification
EI_MAG1	1	
EI_MAG2	2	
EI_MAG3	3	
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_OSABI	7	OS/ABI identification
EI_ABIVERSION	8	ABI version
EI_PAD	9	Start of padding bytes
EI_NIDENT	16	Size of e_ident[]

- e_ident[EI_MAG0] through e_ident[EI_MAG3] contain a “magic number,” identifying the file as an ELF object file. They contain the characters ‘\x7f’, ‘E’, ‘L’, and ‘F’, respectively.

- e_ident[EI_CLASS] identifies the class of the object file, or its capacity. Table 3 lists the possible values.

  This document describes the structures for ELFCLASS64.

  The class of the ELF file is independent of the data model assumed by the object code. The EI_CLASS field identifies the file format; a processor-specific flag in the e_flags field, described below, may be used to identify the application's data model if the processory supports multiple models.

- e_ident[EI_DATA] specifies the data encoding of the object file data structures. Table 4 lists the encodings defined for ELF-64.

  For the convenience of code that examines ELF object files at run time (e.g., the dynamic loader), it is intended that the data encoding of the object file will match that of the running program. For environments that support both byte orders, a processor-specific flag in the e_flags field, described below, may be used to identify the application's operating mode.

- e_ident[EI_VERSION] identifies the version of the object file format. Currently, this field has the value EV_CURRENT, which is defined with the value 1.

- e_ident[EI_OSABI] identifies the operating system and ABI for which the object is prepared. Some fields in other ELF structures have flags and values that have environment-specific meanings; the interpretation of those fields is determined by the value of this field. Table 5 lists the currently-defined values for this field.

- e_ident[EI_ABIVERSION] identifies the version of the ABI for which the object is prepared. This field is used to distinguish among incompatible versions of an ABI. The interpretation of this version number is dependent on the ABI identified by the EI_OSABI field.

  For applications conforming to the System V ABI, third edition, this field should contain 0.

- e_type identifies the object file type. The processor-independent values for this field are listed in Table 6.

- e_machine identifies the target architecture. These values are defined in the processor-specific supplements.

- e_version identifies the version of the object file format. Currently, this field has the value EV_CURRENT, which is defined with the value 1.

- e_entry contains the virtual address of the program entry point. If there is no entry point, this field contains zero.

- e_phoff contains the file offset, in bytes, of the program header table.

- e_shoff contains the file offset, in bytes, of the section header table.

- e_flags contains processor-specific flags.

- e_ehsize contains the size, in bytes, of the ELF header.

- e_phentsize contains the size, in bytes, of a program header table entry.

- e_phnum contains the number of entries in the program header table.

- e_shentsize contains the size, in bytes, of a section header table entry.

- e_shnum contains the number of entries in the section header table.

- e_shstrndx contains the section header table index of the section containing the section name string table. If there is no section name string table, this field has the value SHN_UNDEF.

Table 3. Object File Classes, e_ident[EI_CLASS]

Name	*Value*	*Meaning*
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

Table 4. Data Encodings, e_ident[EI_DATA]

Name	*Value*	*Meaning*
ELFDATA2LSB	1	Object file data structures are little-endian
ELFDATA2MSB	2	Object file data structures are big-endian

Table 5. Operating System and ABI Identifiers, e_ident[EI_OSABI]

Name	*Value*	*Meaning*
ELFOSABI_SYSV	0	System V ABI
ELFOSABI_HPUX	1	HP-UX operating system
ELFOSABI_STANDALONE	255	Standalone (embedded) application

Table 6. Object File Types, e_type

Name	*Value*	*Meaning*
ET_NONE	0	No file type
ET_REL	1	Relocatable object file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOOS	0xFE00	Environment-specific use
ET_HIOS	0xFEFF	
ET_LOPROC	0xFF00	Processor-specific use
ET_HIPROC	0xFFFF	

4. Sections

Sections contain all the information in an ELF file, except for the ELF header, program header table, and section header table. Sections are identified by an index into the section header table.

Section indices

Section index 0, and indices in the range 0xFF00–0xFFFF are reserved for special purposes. Table 7 lists the special section indices that are defined.

Table 7. Special Section Indices

Name	*Value*	*Meaning*
SHN_UNDEF	0	Used to mark an undefined or meaningless section reference
SHN_LOPROC	0xFF00	Processor-specific use
SHN_HIPROC	0xFF1F	
SHN_LOOS	0xFF20	Environment-specific use
SHN_HIOS	0xFF3F	
SHN_ABS	0xFFF1	Indicates that the corresponding reference is an absolute value
SHN_COMMON	0xFFF2	Indicates a symbol that has been declared as a common block (Fortran COMMON or C tentative declaration)

The first entry in the section header table (with an index of 0) is reserved, and must contain all zeroes.

Section header entries

The structure of a section header is shown in Figure 3.

```
typedef struct
{
        Elf64_Word      sh_name;        /* Section name */
        Elf64_Word      sh_type;        /* Section type */
        Elf64_Xword     sh_flags;       /* Section attributes */
        Elf64_Addr      sh_addr;        /* Virtual address in memory */
        Elf64_Off       sh_offset;      /* Offset in file */
        Elf64_Xword     sh_size;        /* Size of section */
        Elf64_Word      sh_link;        /* Link to other section */
        Elf64_Word      sh_info;        /* Miscellaneous information */
        Elf64_Xword     sh_addralign;   /* Address alignment boundary */
        Elf64_Xword     sh_entsize;     /* Size of entries, if section has table */
} Elf64_Shdr;
```

Figure 3. ELF-64 Section Header

- sh_name contains the offset, in bytes, to the section name, relative to the start of the section name string table.
- sh_type identifies the section type. Table 8 lists the processor-independent values for this field.
- sh_flags identifies the attributes of the section. Table 9 lists the processor-independent values for these flags.

- sh_addr contains the virtual address of the beginning of the section in memory. If the section is not allocated to the memory image of the program, this field should be zero.
- sh_offset contains the offset, in bytes, of the beginning of the section contents in the file.
- sh_size contains the size, in bytes, of the section. Except for SHT_NOBITS sections, this is the amount of space occupied in the file.
- sh_link contains the section index of an associated section. This field is used for several purposes, depending on the type of section, as explained in Table 10.
- sh_info contains extra information about the section. This field is used for several purposes, depending on the type of section, as explained in Table 11.
- sh_addralign contains the required alignment of the section. This field must be a power of two.
- sh_entsize contains the size, in bytes, of each entry, for sections that contain fixed-size entries. Otherwise, this field contains zero.

Table 8. Section Types, sh_type

Name	*Value*	*Meaning*
SHT_NULL	0	Marks an unused section header
SHT_PROGBITS	1	Contains information defined by the program
SHT_SYMTAB	2	Contains a linker symbol table
SHT_STRTAB	3	Contains a string table
SHT_RELA	4	Contains “Rela” type relocation entries
SHT_HASH	5	Contains a symbol hash table
SHT_DYNAMIC	6	Contains dynamic linking tables
SHT_NOTE	7	Contains note information
SHT_NOBITS	8	Contains uninitialized space; does not occupy any space in the file
SHT_REL	9	Contains “Rel” type relocation entries
SHT_SHLIB	10	Reserved
SHT_DYNSYM	11	Contains a dynamic loader symbol table
SHT_LOOS	0x60000000	Environment-specific use
SHT_HIOS	0x6FFFFFFF	
SHT_LOPROC	0x70000000	Processor-specific use
SHT_HIPROC	0x7FFFFFFF	

Table 9. Section Attributes, sh_flags

Name	*Value*	*Meaning*
SHF_WRITE	0x1	Section contains writable data
SHF_ALLOC	0x2	Section is allocated in memory image of program
SHF_EXECINSTR	0x4	Section contains executable instructions
SHF_MASKOS	0x0F000000	Environment-specific use
SHF_MASKPROC	0xF0000000	Processor-specific use

Table 10. Use of the sh_link Field

Section Type	*Associated Section*
SHT_DYNAMIC	String table used by entries in this section
SHT_HASH	Symbol table to which the hash table applies
SHT_REL SHT_RELA	Symbol table referenced by relocations
SHT_SYMTAB SHT_DYNSYM	String table used by entries in this section
Other	SHN_UNDEF

Table 11. Use of the sh_info Field

Section Type	*sh_info*
SHT_REL SHT_RELA	Section index of section to which the relocations apply
SHT_SYMTAB SHT_DYNSYM	Index of first non-local symbol (i.e., number of local symbols)
Other	0

Standard sections

The standard sections used for program code and data are shown in Table 12. The standard sections used for other object file information are shown in Table 13. In the flags column, "A" stands for SHF_ALLOC, "W" for SHF_WRITE, and "X" for SHF_EXECINSTR.

Table 12. Standard Sections for Code and Data

Section Name	*Section Type*	*Flags*	*Use*
.bss	SHT_NOBITS	A, W	Uninitialized data
.data	SHT_PROGBITS	A, W	Initialized data
.interp	SHT_PROGBITS	[A]	Program interpreter path name
.rodata	SHT_PROGBITS	A	Read-only data (constants and literals)
.text	SHT_PROGBITS	A, X	Executable code

Table 13. Other Standard Sections

Section Name	*Section Type*	*Flags*	*Use*
.comment	SHT_PROGBITS	none	Version control information
.dynamic	SHT_DYNAMIC	A[, W]	Dynamic linking tables
.dynstr	SHT_STRTAB	A	String table for .dynamic section
.dynsym	SHT_DYNSYM	A	Symbol table for dynamic linking
.got	SHT_PROGBITS	mach. dep.	Global offset table
.hash	SHT_HASH	A	Symbol hash table
.note	SHT_NOTE	none	Note section
.plt	SHT_PROGBITS	mach. dep.	Procedure linkage table
.rel*name* .rela*name*	SHT_REL SHT_RELA	[A]	Relocations for section *name*
.shstrtab	SHT_STRTAB	none	Section name string table
.strtab	SHT_STRTAB	none	String table
.symtab	SHT_SYMTAB	[A]	Linker symbol table

5. String tables

String table sections contain strings used for section names and symbol names. A string table is just an array of bytes containing null-terminated strings. Section header table entries, and symbol table entries refer to strings in a string table with an index relative to the beginning of the string table. The first byte in a string table is defined to be null, so that the index 0 always refers to a null or non-existent name.

6. Symbol table

The first symbol table entry is reserved and must be all zeroes. The symbolic constant STN_UNDEF is used to refer to this entry.

The structure of a symbol table entry is shown in Figure 4.

```
typedef struct
{
        Elf64_Word      st_name;        /* Symbol name */
        unsigned char   st_info;        /* Type and Binding attributes */
        unsigned char   st_other;       /* Reserved */
        Elf64_Half      st_shndx;       /* Section table index */
        Elf64_Addr      st_value;       /* Symbol value */
        Elf64_Xword     st_size;        /* Size of object (e.g., common) */
} Elf64_Sym;
```

Figure 4. ELF-64 Symbol Table Entry

- st_name contains the offset, in bytes, to the symbol name, relative to the start of the symbol string table. If this field contains zero, the symbol has no name.
- st_info contains the symbol type and its binding attributes (that is, its scope). The binding attributes are contained in the high-order four bits of the eight-bit byte, and the symbol type is contained in the low-order four bits. The processor-independent binding attributes are listed in Table 14, and the processor-independent values for symbol type are listed in Table 15.

  An STT_FILE symbol must have STB_LOCAL binding, its section index must be SHN_ABS, and it must precede all other local symbols for the file.
- st_other is reserved for future use; must be zero.
- st_shndx contains the section index of the section in which the symbol is "defined." For undefined symbols, this field contains SHN_UNDEF; for absolute symbols, it contains SHN_ABS; and for common symbols, it contains SHN_COMMON.
- st_value contains the value of the symbol. This may be an absolute value or a relocatable address.

  In relocatable files, this field contains the alignment constraint for common symbols, and a section-relative offset for defined relocatable symbols.

  In executable and shared object files, this field contains a virtual address for defined relocatable symbols.
- st_size contains the size associated with the symbol. If a symbol does not have an associated size, or the size is unknown, this field contains zero.

Table 14. Symbol Bindings

Name	*Value*	*Meaning*
STB_LOCAL	0	Not visible outside the object file
STB_GLOBAL	1	Global symbol, visible to all object files
STB_WEAK	2	Global scope, but with lower precedence than global symbols
STB_LOOS	10	Environment-specific use
STB_HIOS	12	
STB_LOPROC	13	Processor-specific use
STB_HIPROC	15	

Table 15. Symbol Types

Name	*Value*	*Meaning*
STT_NOTYPE	0	No type specified (e.g., an absolute symbol)
STT_OBJECT	1	Data object
STT_FUNC	2	Function entry point
STT_SECTION	3	Symbol is associated with a section

Table 15. Symbol Types (Continued)

Name	*Value*	*Meaning*
STT_FILE	4	Source file associated with the object file
STT_LOOS	10	Environment-specific use
STT_HIOS	12	
STT_LOPROC	13	Processor-specific use
STT_HIPROC	15	

7. Relocations

The ELF format defines two standard relocation formats, "Rel" and "Rela." The first form is shorter, and obtains the addend part of the relocation from the original value of the word being relocated. The second form provides an explicit field for a full-width addend. The structure of relocation entries is shown in Figure 5.

```
typedef struct
{
        Elf64_Addr      r_offset;       /* Address of reference */
        Elf64_Xword     r_info;         /* Symbol index and type of relocation */
} Elf64_Rel;

typedef struct
{
        Elf64_Addr      r_offset;       /* Address of reference */
        Elf64_Xword     r_info;         /* Symbol index and type of relocation */
        Elf64_Sxword    r_addend;       /* Constant part of expression */
} Elf64_Rela;
```

Figure 5. ELF-64 Relocation Entries

- r_offset indicates the location at which the relocation should be applied. For a relocatable file, this is the offset, in bytes, from the beginning of the section to the beginning of the storage unit being relocated. For an executable or shared object, this is the virtual address of the storage unit being relocated.
- r_info contains both a symbol table index and a relocation type. The symbol table index identifies the symbol whose value should be used in the relocation. Relocation types are processor specific. The symbol table index is obtained by applying the ELF64_R_SYM macro to this field, and the relocation type is obtained by applying the ELF64_R_TYPE macro to this field. The ELF64_R_INFO macro combines a symbol table index and a relocation type to produce a value for this field. These macros are defined as follows:

  ```
  #define ELF64_R_SYM(i)((i) > > 32)
  #define ELF64_R_TYPE(i)((i) & 0xf f f f  f f f f L)
  #define ELF64_R_INFO(s, t)(((s) < < 32) + ((t) & 0xf f f f  f f f f L))
  ```

- r_addend specifies a constant addend used to compute the value to be stored in the relocated field.

8. Program header table

In executable and shared object files, sections are grouped into segments for loading. The program header table contains a list of entries describing each segment. The structure of the program header table entry is shown in Figure 6.

```
typedef struct
{
        Elf64_Word      p_type;         /* Type of segment */
        Elf64_Word      p_flags;        /* Segment attributes */
        Elf64_Off       p_offset;       /* Offset in file */
        Elf64_Addr      p_vaddr;        /* Virtual address in memory */
        Elf64_Addr      p_paddr;        /* Reserved */
        Elf64_Xword     p_filesz;       /* Size of segment in file */
        Elf64_Xword     p_memsz;        /* Size of segment in memory */
        Elf64_Xword     p_align;        /* Alignment of segment */
} Elf64_Phdr;
```

Figure 6. ELF-64 Program Header Table Entry

- p_type identifies the type of segment. The processor-independent segment types are shown in Table 16.
- p_flags contains the segment attributes. The processor-independent flags are shown in Table 17. The top eight bits are reserved for processor-specific use, and the next eight bits are reserved for environment-specific use.
- p_offset contains the offset, in bytes, of the segment from the beginning of the file.
- p_vaddr contains the virtual address of the segment in memory.
- p_paddr is reserved for systems with physical addressing.
- p_filesz contains the size, in bytes, of the file image of the segment.
- p_memsz contains the size, in bytes, of the memory image of the segment.
- p_align specifies the alignment constraint for the segment. Must be a power of two. The values of p_offset and p_vaddr must be congruent modulo the alignment.

Table 16. Segment Types, p_type

Name	*Value*	*Meaning*
PT_NULL	0	Unused entry
PT_LOAD	1	Loadable segment
PT_DYNAMIC	2	Dynamic linking tables
PT_INTERP	3	Program interpreter path name
PT_NOTE	4	Note sections

Table 16. Segment Types, p_type (Continued)

Name	*Value*	*Meaning*
PT_SHLIB	5	Reserved
PT_PHDR	6	Program header table
PT_LOOS	0x60000000	Environment-specific use
PT_HIOS	0x6FFFFFFF	
PT_LOPROC	0x70000000	Processor-specific use
PT_HIPROC	0x7FFFFFFF	

Table 17. Segment Attributes, p_flags

Name	*Value*	*Meaning*
PF_X	0x1	Execute permission
PF_W	0x2	Write permission
PF_R	0x4	Read permission
PF_MASKOS	0x00FF0000	These flag bits are reserved for environment-specific use
PF_MASKPROC	0xFF000000	These flag bits are reserved for processor-specific use

9. Note sections

Sections of type SHT_NOTE and segments of type PT_NOTE are used by compilers and other tools to mark an object file with special information that has special meaning to a particular tool set. These sections and segments contain any number of note entries, each of which is an array of 8-byte words in the byte order defined in the ELF file header. The format of a note entry is shown in Figure 7.

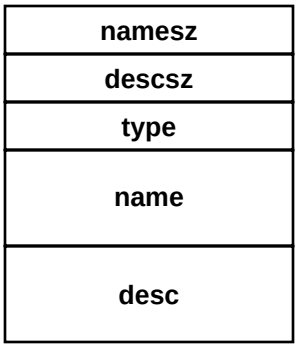

Figure 7. Format of a Note Section

- namesz and name The first word in the entry, namesz, identifies the length, in bytes, of a name identifying the entry's owner or originator. The name field contains a null-terminated string, with padding as necessary to ensure 8-

byte alignment for the descriptor field. The length does not include the terminating null or the padding. By convention, each vendor should use its own name in this field.

- descsz and desc The second word in the entry, descsz, identifies the length of the note descriptor. The desc field contains the contents of the note, followed by padding as necessary to ensure 8-byte alignment for the next note entry. The format and interpretation of the note contents are determined solely by the name and type fields, and are unspecified by the ELF standard.
- type The third word contains a number that determines, along with the originator's name, the interpretation of the note contents. Each originator controls its own types.

10. Dynamic table

Dynamically-bound object files will have a PT_DYNAMIC program header entry. This program header entry refers to a segment containing the .dynamic section, whose contents are an array of Elf64_Dyn structures. The dynamic structure is shown in Figure 8.

```
typedef struct
{
        Elf64_Sxword        d_tag;
        union {
                Elf64_Xword         d_val;
                Elf64_Addr          d_ptr;
        } d_un;
} Elf64_Dyn;

extern Elf64_Dyn _DYNAMIC[];
```

Figure 8. Dynamic Table Structure

- d_tag Identifies the type of dynamic table entry. The type determines the interpretation of the d_un union. The processor-independent dynamic table entry types are shown in Table 18. Other values, in the range 0x7000 0000–0x7FFF FFFF, may be defined as processor-specific types.

  String table offsets are relative to the beginning of the table identified by the DT_STRTAB entry. All strings in the string table must be null-terminated.
- d_val This union member is used to represent integer values.
- d_ptr This union member is used to represent program virtual addresses. These addresses are link-time virtual addresses, and must be relocated to match the object file's actual load address. This relocation must be done implicitly; there are no dynamic relocations for these entries.

Table 18. Dynamic Table Entries

Name	*Value*	*d_un*	*Meaning*
DT_NULL	0	*ignored*	Marks the end of the dynamic array
DT_NEEDED	1	d_val	The string table offset of the name of a needed library.

Table 18. Dynamic Table Entries (Continued)

Name	*Value*	*d_un*	*Meaning*
DT_PLTRELSZ	2	d_val	Total size, in bytes, of the relocation entries associated with the procedure linkage table.
DT_PLTGOT	3	d_ptr	Contains an address associated with the linkage table. The specific meaning of this field is processor-dependent.
DT_HASH	4	d_ptr	Address of the symbol hash table, described below.
DT_STRTAB	5	d_ptr	Address of the dynamic string table.
DT_SYMTAB	6	d_ptr	Address of the dynamic symbol table.
DT_RELA	7	d_ptr	Address of a relocation table with Elf64_Rela entries.
DT_RELASZ	8	d_val	Total size, in bytes, of the DT_RELA relocation table.
DT_RELAENT	9	d_val	Size, in bytes, of each DT_RELA relocation entry.
DT_STRSZ	10	d_val	Total size, in bytes, of the string table.
DT_SYMENT	11	d_val	Size, in bytes, of each symbol table entry.
DT_INIT	12	d_ptr	Address of the initialization function.
DT_FINI	13	d_ptr	Address of the termination function.
DT_SONAME	14	d_val	The string table offset of the name of this shared object.
DT_RPATH	15	d_val	The string table offset of a shared library search path string.
DT_SYMBOLIC	16	*ignored*	The presence of this dynamic table entry modifies the symbol resolution algorithm for references within the library. Symbols defined within the library are used to resolve references before the dynamic linker searches the usual search path.
DT_REL	17	d_ptr	Address of a relocation table with Elf64_Rel entries.
DT_RELSZ	18	d_val	Total size, in bytes, of the DT_REL relocation table.
DT_RELENT	19	d_val	Size, in bytes, of each DT_REL relocation entry.
DT_PLTREL	20	d_val	Type of relocation entry used for the procedure linkage table. The d_val member contains either DT_REL or DT_RELA.
DT_DEBUG	21	d_ptr	Reserved for debugger use.
DT_TEXTREL	22	*ignored*	The presence of this dynamic table entry signals that the relocation table contains relocations for a non-writable segment.
DT_JMPREL	23	d_ptr	Address of the relocations associated with the procedure linkage table.
DT_BIND_NOW	24	*ignored*	The presence of this dynamic table entry signals that the dynamic loader should process all relocations for this object before transferring control to the program.
DT_INIT_ARRAY	25	d_ptr	Pointer to an array of pointers to initialization functions.
DT_FINI_ARRAY	26	d_ptr	Pointer to an array of pointers to termination functions.
DT_INIT_ARRAYSZ	27	d_val	Size, in bytes, of the array of initialization functions.
DT_FINI_ARRAYSZ	28	d_val	Size, in bytes, of the array of termination functions.
DT_LOOS	0x60000000		Defines a range of dynamic table tags that are reserved for environment-specific use.

Table 18. Dynamic Table Entries (Continued)

Name	*Value*	*d_un*	*Meaning*
DT_HIOS	0x6FFFFFFF		
DT_LOPROC	0x70000000		Defines a range of dynamic table tags that are reserved for processor-specific use.
DT_HIPROC	0x7FFFFFFF		

11. Hash table

The dynamic symbol table can be accessed efficiently through the use of a hash table. The hash table is part of a loaded program segment, typically in the .hash section, and is pointed to by the DT_HASH entry in the dynamic table. The hash table is an array of Elf64_Word objects, organized as shown in Figure 9.

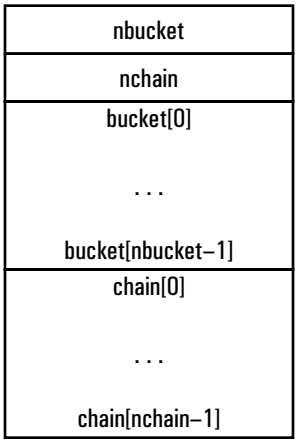

Figure 9. Symbol Hash Table

The bucket array forms the hash table itself. The number of entries in the hash table is given by the first word, nbucket, and may be chosen arbitrarily.

The entries in the chain array parallel the symbol table—there is one entry in the chain table for each symbol in the symbol table, so nchain should equal the number of symbol table entries.

Symbols in the symbol table are organized into hash chains, one chain per bucket. A hash function, shown in Figure 10, computes a hash value x for a given symbol name. The value of bucket[x % nbucket] is the symbol table index for the first symbol on the hash chain. The index next symbol on the hash chain is given by the entry in the chain array with the same index. The hash chain can be followed until a chain array entry equal to STN_UNDEF is found, marking the end of the chain.

```
unsigned long
elf64_hash(const unsigned char *name)
{
        unsigned long h = 0, g;

        while (*name) {
                h = (h << 4) + *name++;
                if (g = h & 0xf0000000)
                        h ^= g >> 24;
                h &= 0x0fffffff;
        }
        return h;
}
```

Figure 10. Hash Function