ELF-64 Object File Format

Including HP and HP-UX Extensions

Version 1.4
May 20, 1997

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. It also contains the HP vendor-specific and HP-UX environment-specific extensions to ELF.

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.

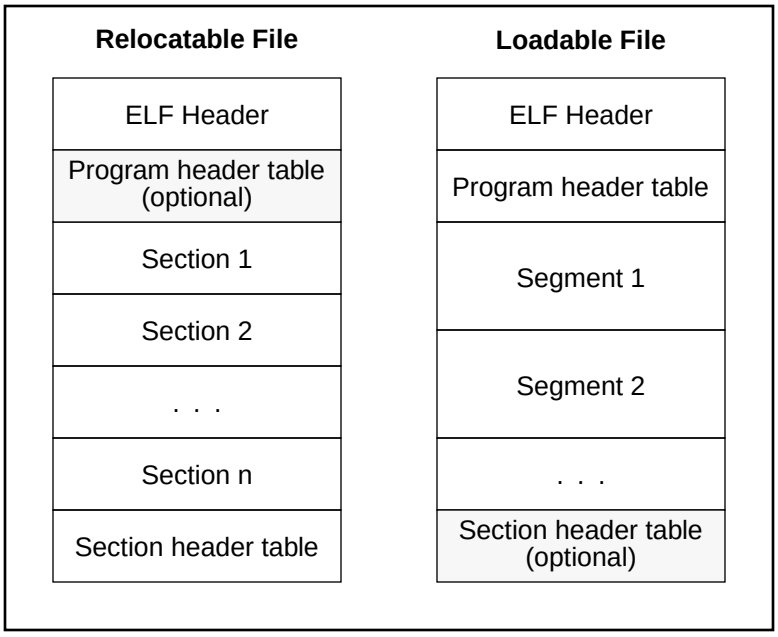

Figure 1. Structure of an ELF file

2. Data representation

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

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 — These 16 bytes 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 following table:

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]
These bytes 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. The following table shows the possible values:

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

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.

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

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. The following table shows the currently-defined values for this field:

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

This document includes extensions for the HP-UX operating system. All material labeled as an HP or HP-UX extension is applicable only to object files marked with ELFOSABI_HPUX in 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.

Applications conforming to the HP-UX ABI defined for HP-UX Release 11.0 should have the value 1 in this field. A value of 0 in this field indicates that no version is specified, and no compatibility checking should be applied to the object module.

e_type — Identifies the object file type. The processor-independent values for this field are shown in the following table.

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	Defines a range of object file types that is reserved for environment-specific use
ET_HIOS	0xFEFF	
ET_LOPROC	0xFF00	Defines a range of object file types that is reserved for processor-specific use
ET_HIPROC	0xFFFF	

One object file type is specific to the HP-UX operating environment. It is shown in the following table.

Name	Value	Meaning
ET_HP_IFILE	0xFE00	Intermediate object code

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

e_phoff — The file offset, in bytes, of the program header table.

e_shoff — The file offset, in bytes, of the section header table.

e_flags — Processor-specific flags.

e_ehsize — Size, in bytes, of the ELF header.

e_phentsize — Size, in bytes, of a program header table entry.

e_phnum — Number of entries in the program header table.

e_shentsize — Size, in bytes, of a section header table entry.

e_shnum — Number of entries in the section header table.

e_shstrndx — 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.

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. The following special indices are currently defined:

Name	Value	Meaning
SHN_UNDEF	0	Used to mark an undefined or meaningless section reference
SHN_LOPROC	0xFF00	Defines a range of section indices that is reserved for processor-specific use
SHN_HIPROC	0xFF1F	
SHN_LOOS	0xFF20	Defines a range of section indices that is reserved for 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.

One special section index is specific to the HP-UX operating environment. It is shown in the following table.

Name	Value	Meaning
SHN_TLS_COMMON	0xFF20	Indicates a symbol that has been declared as a common block in thread-local storage (Fortran COMMON or C tentative declaration)

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 — Offset, in bytes, to the section name, relative to the start of the section name string table.

sh_type — Identifies the section type. The processor-independent values for this field are shown in the following table.

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	Defines a range of section types that is reserved for environment-specific use
SHT_HIOS	0x6FFFFFFF	

Name	Value	Meaning
SHT_LOPROC	0x70000000	Defines a range of section types that is reserved for processor-specific use
SHT_HIPROC	0x7FFFFFFF	

The section types specific to the HP-UX operating environment are shown in the following table.

Name	Value	Meaning
SHT_HP_OVLBITS	0x60000000	Contains information defined by the program that should overlay other instances of the same section
SHT_HP_DLKM	0x60000001	Contains information for dynamically-loaded kernel modules
SHT_HP_COMDAT	0x60000002	Contains a directory of sections that may be duplicated in other object files

sh_flags

Identifies the attributes of the section. The processor-independent values for these flags are shown in the following table.

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	These bits are reserved for environment-specific use
SHF_MASKPROC	0xF0000000	These bits are reserved for processor-specific use

Section attributes specific to the HP-UX operating environment are shown in the following table.

Name	Value	Meaning
SHF_HP_TLS	0x01000000	Section contains thread-local storage
SHF_HP_NEAR_SHARED	0x02000000	Section contains near-shared data
SHF_HP_FAR_SHARED	0x04000000	Section contains far-shared data
SHF_HP_COMDAT	0x08000000	Section is a member of a comdat group

sh_addr

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

Offset, in bytes, of the beginning of the section contents in the file.

sh_size — 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 the following table.

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

sh_info — Contains extra information about the section. This field is used for several purposes, depending on the type of section, as explained in the following table.

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

For SHT_HP_COMDAT and SHT_HP_OVLBITS sections, this field contains selection and merge criteria for the section. See “Comdat section groups” and “Overlay sections” below.

sh_addralign — Alignment required. This field must be a power of two.

sh_entsize — For sections that contain fixed-size entries, this field contains the size, in bytes, of each entry. Otherwise, this field contains zero.

Standard sections

The standard sections used for program code and data are shown in the following table. In the flags column, “A” stands for SHF_ALLOC, “W” for SHF_WRITE, and “X” for SHF_EXECINSTR.

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

Sections that are specific to the HP-UX operating system are shown in the following table. In the "Flags" column, "T" stands for SHF_HP_TLS, "NS" for SHF_HP_NEAR_SHARED, and "FS" for SHF_HP_FAR_SHARED.

Section Name	Section Type	Flags	Use
.fini	SHT_PROGBITS	A, W	Process termination code
.init	SHT_PROGBITS	A, W	Process initialization code
.preinit	SHT_PROGBITS	A, W	Process pre-initialization code
.tbss	SHT_NOBITS	A, W, T	Uninitialized thread-local storage
.HP.nsdata	SHT_PROGBITS	A, W, NS	Initialized near-shared data
.HP.nsbss	SHT_NOBITS	A, W, NS	Uninitialized near-shared data
.HP.fsdata	SHT_PROGBITS	A, W, FS	Initialized far-shared data
.HP.fsbss	SHT_NOBITS	A, W, FS	Uninitialized far-shared data

The standard sections used for other object file information are shown in the following table.

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

Comdat section groups

A comdat (short for "common data") section group is a group of sections in an object file that may be duplicated in one or more other object files. The linker is expected to keep one group and ignore all others. A comdat group consists of one SHT_HP_COMDAT section and one or more member sections. Global symbols defined relative to any ignored sections will be treated as unsatisfied symbols; local symbols will be discarded.

The SHT_HP_COMDAT section serves as a directory of the member sections, which allows the linker to associate the members of the group with one another, and with a key that identifies each group. The name of the SHT_HP_COMDAT section serves as the key. Members of a comdat group are ordinary sections, except that each member section must have the SHF_HP_COMDAT attribute set in its section header entry.

The section header entry for the SHT_HP_COMDAT section must be constructed as follows:

sh_name — Offset, in bytes, to the identifier for the comdat group, relative to the start of the section name string table.

sh_type — This field must contain the value SHT_HP_COMDAT.

sh_flags — This field must be set to 0.

sh_addr — This field is not used, and must be set to 0.

sh_offset — Offset, in bytes, to the beginning of the directory, relative to the start of the file header.

sh_size — Size, in bytes, of the directory.

sh_link — This field is not used, and must be set to 0.

sh_info — The valid selection criteria for a comdat section are shown in the following table.

Name	Value	Meaning
SHI_HP_PICKANY	1	Pick any one instance of this section group

sh_addralign — This field is not used, and must be set to 0.

sh_entsize — Size, in bytes, of each entry in the directory. This value is currently 4 (the size of an Elf64_Word).

The comdat group directory is an array of section indices. Each entry is an Elf64_Word object, whose value is the section index of a member section of the comdat group.

Overlay sections

An overlay section contains data that is meant to overlay the contents of other sections with the same section name. Each input section in an overlay is assigned the same starting address; the size of the combined output section is the equal to the largest of the input sections in the overlay. The method for combining the data from each input section is dependent on the selection criteria, described below.

The fields of an overlay section header entry have the same interpretation as those for an SHT_PROGBITS section, with the following exception:

sh_info — The valid selection criteria for an overlay section are shown in the following table.

Name	Value	Meaning
SHI_HP_MERGE	0	Merge all instances of this section together; transparent regions within one section will not overwrite opaque regions of another section

When overlay sections use the SHI_HP_MERGE method of combining contents, the contents of each input section are divided into one or more regions, each one of which is designated “opaque” or “transparent.” A transparent region will not overwrite the corresponding region of any other section in the same overlay. If part of the combined output section is covered by opaque regions from more than one input section, the linker will select one of the opaque regions arbitrarily. Relocations for all input sections may be processed; at a minimum, relocations corresponding to each selected opaque region will be processed. It is the compiler's responsibility to ensure that the application of multiple relocations for the same location will not cause incorrect behavior.

The contents of an overlay section with SHI_HP_MERGE selection are considered transparent by default; opaque regions are designated by the use of special STT_HP_OPAQUE symbols, described in Section 6. The section contents in the object file must include the transparent regions; these regions should be filled with zeroes. The linker, however, will ignore the transparent regions, and read only the opaque regions.

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. This index has a defined name:

Name	Value
STN_UNDEF	0

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 — 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 — This field 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 shown in the following table.

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

Name	Value	Meaning
STB_LOOS	10	Defines a range of binding attributes that is reserved for environment-specific use
STB_HIOS	12	
STB_LOPROC	13	Defines a range of binding attributes that is reserved for processor-specific use
STB_HIPROC	15	

The processor-independent values for symbol type are shown in the following table.

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
STT_FILE	4	Source file associated with the object file
STT_LOOS	10	Defines a range of symbol types that is reserved for environment-specific use
STT_HIOS	12	
STT_LOPROC	13	Defines a range of symbol types that is reserved for processor-specific use
STT_HIPROC	15	

The following table shows the symbol type defined for the HP-UX operating system.

Name	Value	Meaning
STT_HP_OPAQUE	11	Designates an opaque region within an overlay section
STT_HP_STUB	12	Identifies the beginning of a contiguous region of stubs.

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.

An STT_HP_OPAQUE symbol designates an opaque region within an overlay section (see “Overlay sections” in Section 4). The symbol value and size are used to identify the region. The symbol name is ignored; it may be null. The symbol should have STB_LOCAL binding. An overlay section may have any number of opaque symbols, and the regions designated by multiple symbols may overlap. All STT_HP_OPAQUE symbols are discarded during a final link.

st_other — Reserved for future use; must be zero.

st_shndx — 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 — Size associated with the symbol. If a symbol does not have an associated size, or the size is unknown, this field contains zero.

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) & 0xffff ffffL)
#define ELF64_R_INFO(s, t)      (((s) << 32) + ((t) & 0xffff ffffL))
```

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 the following table.

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
PT_SHLIB	5	Reserved
PT_PHDR	6	Program header table
PT_LOOS	0x60000000	Defines a range of segment types that is reserved for environment-specific use
PT_HIOS	0x6FFFFFFF	
PT_LOPROC	0x70000000	Defines a range of segment types that is reserved for processor-specific use
PT_HIPROC	0x7FFFFFFF	

Segment types that are specific to the HP-UX operating system are shown in the following table.

Name	Value	Meaning
PT_HP_TLS	0x60000000	Segment contains thread-local storage

Name	Value	Meaning
PT_HP_CORE_NONE	0x60000001	Segment is part of a core file
PT_HP_CORE_VERSION	0x60000002	
PT_HP_CORE_KERNEL	0x60000003	
PT_HP_CORE_COMM	0x60000004	
PT_HP_CORE_PROC	0x60000005	
PT_HP_CORE_LOADABLE	0x60000006	
PT_HP_CORE_STACK	0x60000007	
PT_HP_CORE_SHM	0x60000008	
PT_HP_CORE_MMF	0x60000009	
PT_HP_PARALLEL	0x60000010	Segment contains parallel program header
PT_HP_FASTBIND	0x60000011	Segment contains fastbind information

p_flags

Segment attributes. The processor-independent flags are shown in the following table. The top eight bits are reserved for processor-specific use, and the next eight bits are reserved for environment-specific use.

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

Segment attributes that are specific to the HP-UX operating system are shown in the following table.

Name	Value	Meaning
PF_HP_LAZYSWAP	0x00800000	Lazy-swap allocation allowed for this segment
PF_HP_NEAR_SHARED	0x00400000	Segment should be allocated in near-shared memory
PF_HP_FAR_SHARED	0x00200000	Segment should be allocated in far-shared memory
PF_HP_PAGE_SIZE	0x00100000	Segment should be mapped with page size specified in p_align field
PF_HP_MODIFY	0x00080000	Segment contains writable data that is expected to be modified
PF_HP_CODE	0x00040000	Segment contains executable code

p_offset

Offset, in bytes, of the segment from the beginning of the file.

p_vaddr — Virtual address of the segment in memory.

p_paddr — Reserved for systems with physical addressing.

p_filesz — Size, in bytes, of the file image of the segment.

p_memsz — Size, in bytes, of the memory image of the segment.

p_align — 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.

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.

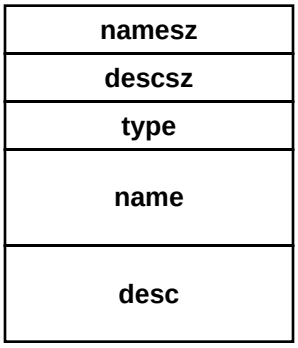

Figure 7. Format of a Note Section

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.

HP compilers define several note section types. Each note section is identified by the name “HP” and one of the types shown in the following table. In all cases, the note descriptor contains a null-terminated string.

Name	Value	Contents
NOTE_HP_COMPILER	1	Compiler identification string
NOTE_HP_COPYRIGHT	2	Copyright string
NOTE_HP_VERSION	3	Version string

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 the following table. Other values, in the range 0x7000 0000–0x7FFF FFFF, may be defined as processor-specific types.

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.
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.

Name	Value	d_un	Meaning
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.

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

The following table shows the dynamic table entries that are specific to the HP-UX operating system:

Name	Value	d_un	Meaning
DT_HP_LOAD_MAP	0x60000000	d_ptr	Address of a location in the module's data segment where the dynamic linker can store the address of its load map, for use in cross-module callbacks.
DT_HP_DLD_FLAGS	0x60000001	d_val	Flags for use with the debugger. These are described in a separate table below.

Name	Value	d_un	Meaning
DT_HP_DLD_HOOK	0x60000002	d_ptr	Address of a function pointer that refers to a callback routine that the dynamic linker should call when modules are loaded and unloaded.
DT_HP_UX10_INIT	0x60000003	d_val	Address of the HP-UX 10.x-compatible initializer list
DT_HP_UX10_INITSZ	0x60000004	d_val	Size, in bytes, of the HP-UX 10.x-compatible initializer list
DT_HP_PREINIT	0x60000005	d_val	Address of the .preinit section
DT_HP_PREINITSZ	0x60000006	d_val	Total size, in bytes, of the .preinit section
DT_HP_NEEDED	0x60000007	d_val	The string table offset of the name of a needed library (HP-UX 10.x compatibility)
DT_HP_TIME_STAMP	0x60000008	d_val	Time stamp for fastbind information
DT_HP_CHECKSUM	0x60000009	d_val	Checksum for fastbind information

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.

The flags that can be specified with the DT_HP_DLD_FLAGS entry are shown in the following table:

Name	Value	Meaning
DT_HP_DEBUG_PRIVATE	0x00000001	Specifies that all DLLs should be mapped with their text segments in private memory.
DT_HP_DEBUG_CALLBACK	0x00000002	Specifies that a callback routine should be invoked for all load and unload events. The callback routine is identified by the DT_HP_DLD_HOOK entry.
DT_HP_DEBUG_CALLBACK_BOR	0x00000004	Specifies that a callback routine should be invoked for all bind-on-reference events. The callback routine is identified by the DT_HP_DLD_HOOK entry.
DT_HP_NO_ENVVAR	0x00000008	Specifies that no environment variable should be used in the search for any DLL in the process. If this is set, only the embedded RPATH values and default directory locations will be used.

Name	Value	Meaning
DT_HP_BIND_NOW	0x00000010	Specifies that all external references from this load module should be bound immediately
DT_HP_BIND_NONFATAL	0x00000020	Unresolved references should be treated as warnings
DT_HP_BIND_VERBOSE	0x00000040	Dynamic loader will print binding information
DT_HP_BIND_RESTRICTED	0x00000080	Deferred bindings will be restricted to those libraries that were available when this load module was loaded
DT_HP_BIND_SYMBOLIC	0x00000100	External references should be bound internally if possible
DT_HP_BIND_RPATH_FIRST	0x00000200	The list of libraries designated by the embedded RPATH values should be searched prior to those specified by a run-time environment variable
DT_HP_BIND_DEPTH_FIRST	0x00000400	External references should be resolved with a depth-first search instead of a breadth-first search

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.

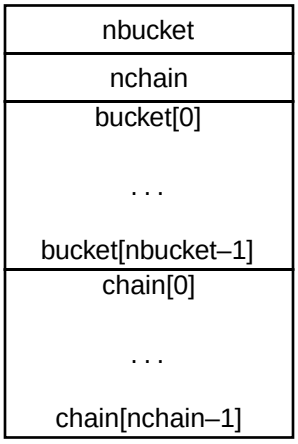

Figure 9. Symbol Hash Table

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