Symbol Table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified later in this section.

Name	Value
STN_UNDEF	0

A symbol table entry has the following format.

Figure 1-16: Symbol Table Entry

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;
```

`st_name` This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

NOTE External C symbols have the same names in C and object files' symbol tables.

`st_value` This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, etc.; details appear below.

`st_size` Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.

`st_info` This member specifies the symbol's type and binding attributes. A list of the values and meanings appears below. The following code shows how to manipulate the values.

```
#define ELF32_ST_BIND(i)   ((i)>>4)
#define ELF32_ST_TYPE(i)   ((i)&0xf)
#define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))
```

`st_other` This member currently holds 0 and has no defined meaning.

`st_shndx` Every symbol table entry is ''defined'' in relation to some section; this member holds the relevant section header table index. As Figure 1-7 and the related text describe, some section indexes indicate special meanings.

A symbol's binding determines the linkage visibility and behavior.

Figure 1-17: Symbol Binding, `ELF32_ST_BIND`

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOPROC	13
STB_HIPROC	15

`STB_LOCAL` Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.

`STB_GLOBAL` Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same global symbol.

`STB_WEAK` Weak symbols resemble global symbols, but their definitions have lower precedence.

`STB_LOPROC` through `STB_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics.

Global and weak symbols differ in two major ways.

- When the link editor combines several relocatable object files, it does not allow multiple definitions of `STB_GLOBAL` symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error. The link editor honors the global definition and ignores the weak ones. Similarly, if a common symbol exists (i.e., a symbol whose st_shndx field holds `SHN_COMMON`), the appearance of a weak symbol with the same name will not cause an error. The link editor honors the common definition and ignores the weak ones.
- When the link editor searches archive libraries, it extracts archive members that contain definitions of undefined global symbols. The member's definition may be either a global or a weak symbol. The link editor does *not* extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value.

In each symbol table, all symbols with `STB_LOCAL` binding precede the weak and global symbols. As ''Sections'' above describes, a symbol table section's `sh_info` section header member holds the symbol table index for the first non-local symbol.

A symbol's type provides a general classification for the associated entity.

Figure 1-18: Symbol Types, `ELF32_ST_TYPE`

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
STT_HIPROC	15

`STT_NOTYPE` The symbol's type is not specified.

`STT_OBJECT` The symbol is associated with a data object, such as a variable, an array, etc.

`STT_FUNC` The symbol is associated with a function or other executable code.

`STT_SECTION` The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have `STB_LOCAL` binding.

`STT_FILE` Conventionally, the symbol's name gives the name of the source file associated with the object file. A file symbol has `STB_LOCAL` binding, its section index is `SHN_ABS`, and it precedes the other `STB_LOCAL` symbols for the file, if it is present.

`STT_LOPROC` through `STT_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics.

Function symbols (those with type `STT_FUNC`) in shared object files have special significance. 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. Shared object symbols with types other than `STT_FUNC` will not be referenced automatically through the procedure linkage table.

If a symbol's value refers to a specific location within a section, its section index member, `st_shndx`, holds an index into the section header table. As the section moves during relocation, the symbol's value changes as well, and references to the symbol continue to ''point'' to the same location in the program. Some special section index values give other semantics.

`SHN_ABS` The symbol has an absolute value that will not change because of relocation.

`SHN_COMMON` The symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's `sh_addralign` member. That is, the link editor will allocate the storage for the symbol at an address that is a multiple of `st_value`. The symbol's size tells how many bytes are required.

`SHN_UNDEF` This section table index means the symbol is undefined. When the link editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition.

As mentioned above, the symbol table entry for index 0 (STN_UNDEF) is reserved; it holds the following.

Figure 1-19: Symbol Table Entry: Index 0

Name	Value	Note
st_name	0	No name
st_value	0	Zero value
st_size	0	No size
st_info	0	No type, local binding
st_other	0	
st_shndx	SHN_UNDEF	No section

Symbol Values

Symbol table entries for different object file types have slightly different interpretations for the st_value member.

- In relocatable files, st_value holds alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, st_value holds a section offset for a defined symbol. That is, st_value is an offset from the beginning of the section that st_shndx identifies.
- In executable and shared object files, st_value holds a virtual address. To make these files' symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different object files, the data allow efficient access by the appropriate programs.