11. COMMON OBJECT FILE FORMAT (COFF)

The Common Object File Format (COFF)

This section describes the Common Object File Format (COFF), the format of the output file produced by the assembler, **as**, and the link editor, **ld**.

Some key features of COFF are:

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications
- programmers can modify the way the object file is constructed by providing directives at compile time

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains:

- a file header
- optional header information
- a table of section headers
- data corresponding to the section headers
- relocation information
- line numbers
- a symbol table
- a string table

Figure 11-1 shows the overall structure.

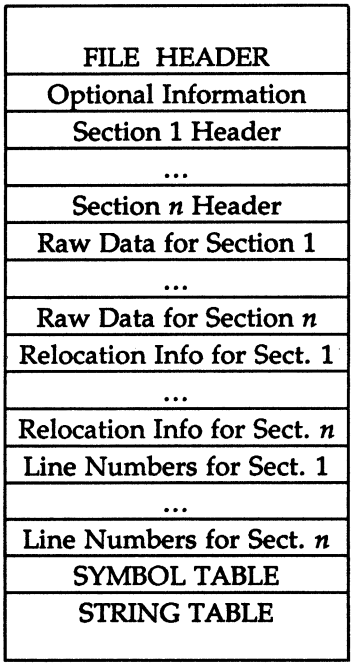


Figure 11-1. Object File Format

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **–s** option of the **ld** command or if the line number information, symbol table, and string table are removed by the **strip** command. The line number information does not appear unless the program is compiled with the **–g** option of the **cc** command. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references is considered executable.

Definitions and Conventions

Before proceeding further, you should become familiar with the following terms and conventions.

Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate comments, multiple text or data segments, shared data segments, or user-specified sections. However, the operating system loads only **.text**, **.data**, and **.bss** into memory when the file is executed.

NOTE

It is a mistake to assume that every COFF file will have a specific number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain it from file and section headers in the file.

Physical and Virtual Addresses

The physical address of a section or symbol is the offset of that section or symbol from address zero of the address space. The term physical address as used in COFF does not correspond to general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section header contains two address fields, a physical address, and a virtual address; but in all versions of COFF, the physical address is equivalent to the virtual address.

Target Machine

Compilers and link editors produce executable object files that are intended to be run on a particular computer. In the case of cross-compilers, the compilation and link editing are done on one computer with the intent of creating an object file that can be executed on another computer. The term target machine refers to the computer on which the object file is destined to run. Usually, the target machine is the same computer on which the object file is being created.

File Header

The file header contains the 20 bytes of information shown in Figure 11-2. The last two bytes are flags that are used by **ld** and object file utilities.

Bytes	Declaration	Name	Description
0-1	**unsigned short**	**f_magic**	Magic number
2-3	**unsigned short**	**f_nscns**	Number of sections
4-7	**long int**	**f_timdat**	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	**long int**	**f_symptr**	File pointer containing the starting address of the symbol table
12-15	**long int**	**f_nsyms**	Number of entries in the symbol table
16-17	**unsigned short**	**f_opthdr**	Number of bytes in the optional header
18-19	**unsigned short**	**f_flags**	Flags (see Figure 11-3)

Figure 11-2. File Header Contents

Magic Numbers

The magic number specifies the target machine on which the object file is executable.

Flags

The last two bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file **filehdr.h**, and are shown in Figure 11-3.

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_MINMAL	00020	Not used by SYSTEM V/68
F_UPDATE	00040	Not used by SYSTEM V/68
F_SWABD	00100	Not used by SYSTEM V/68
F_AR16WR	00200	File has the byte ordering used by the PDP-11/70 processor
F_AR32WR	00400	File has the byte ordering used by the VAX-11/780 (i.e., 32 bits per word, least significant byte first)
F_AR32W	01000	File has the byte ordering used by the M68K computers (i.e., 32 bits per word, most significant byte first)
F_PATCH	02000	Not used by SYSTEM V/68

Figure 11-3. File Header Flags

File Header Declaration

The C structure declaration for the file header is given in Figure 11-4. This declaration may be found in the header file **filehdr.h**.

```
struct filehdr
{
    unsigned short   f_magic;   /* magic number */
    unsigned short   f_nscns;   /* number of section */

    long             f_timdat;  /* time and date stamp */

    long             f_symptr;  /* file ptr to symbol table */

    long             f_nsyms;   /* number entries in the symbol table */

    unsigned short   f_opthdr;  /* size of optional header */

    unsigned short   f_flags;   /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)
```

Figure 11-4. File Header Declaration

Optional Header Information

The template for optional information varies among different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions, the disassembler, etc.) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header field **f_opthdr**.

Standard Operating System a.out Header

By default, files produced by the link editor for the operating system always have a standard operating system **a.out** header in the optional header field. The operating system **a.out** header is 28 bytes. The fields of the optional header are described in Figure 11-5.

Bytes	Declaration	Name	Description
0-1	**short**	**magic**	Magic number
2-3	**short**	**vstamp**	Version stamp
4-7	**long int**	**tsize**	Size of text in bytes
8-11	**long int**	**dsize**	Size of initialized data in bytes
12-15	**long int**	**bsize**	Size of uninitialized data in bytes
16-19	**long int**	**entry**	Entry point
20-23	**long int**	**text_start**	Base address of text
24-27	**long int**	**data_start**	Base address of data

Figure 11-5. Optional Header Contents

Whereas, the magic number in the file header specifies the machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by the operating system are given in Figure 11-6.

Value	Meaning
0407	The text segment is not write-protected or sharable; the data segment is contiguous with the text segment.
0410	The data segment starts at the next segment following the text segment and the text segment is write protected.
0413	Text and data segments are aligned within **a.out** so it can be directly paged.

Figure 11-6. Operating System Magic Numbers

Optional Header Declaration

The C language structure declaration currently used for the operating system **a.out** file header is given in Figure 11-7. This declaration may be found in the header file **aouthdr.h**.

```
typedef struct aouthdr
{
        short    magic;         /* magic number */
        short    vstamp;        /* version stamp */
        long     tsize;         /* text size in bytes, padded */

                                /* to full word boundary */

        long     dsize;         /* initialized data size */

        long     bsize;         /* uninitialized data size */

        long     entry;         /* entry point */

        long     text_start;    /* base of text for this file */

        long     data_start     /* base of data for this file */

} AOUTHDR;
```

Figure 11-7. aouthdr Declaration

Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Figure 11-8.

Bytes	Declaration	Name	Description
0-7	**char**	**s_name**	8-character null padded section name
8-11	**long int**	**s_paddr**	Physical address of section
12-15	**long int**	**s_vaddr**	Virtual address of section
16-19	**long int**	**s_size**	Section size in bytes
20-23	**long int**	**s_scnptr**	File pointer to raw data
24-27	**long int**	**s_relptr**	File pointer to relocation entries
28-31	**long int**	**s_lnnoptr**	File pointer to line number entries
32-33	**unsigned short**	**s_nreloc**	Number of relocation entries
34-35	**unsigned short**	**s_nlnno**	Number of line number entries
36-39	**long int**	**s_flags**	Flags (see Figure 11-9)

Figure 11-8. Section Header Contents

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the operating system function **fseek**(3S).

Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Figure 11-9.

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data
STYP_INFO	0x200	Comment section (not allocated, not relocated, not loaded)
STYP_OVER	0x400	Overlay section (relocated, not allocated, not loaded)
STYP_LIB	0x800	For **.lib** section (treated like STYP_INFO)

Figure 11-9. Section Header Flags

Section Header Declaration

The C structure declaration for the section headers is described in Figure 11-10. This declaration may be found in the header file **scnhdr.h**.

```
struct scnhdr
{
        char        s_name[8];          /* section name */
        long        s_paddr;            /* physical address */
        long        s_vaddr;            /* virtual address */
        long        s_size;             /* section size */
        long        s_scnptr;           /* file ptr to section raw data */

        long        s_relptr;           /* file ptr to relocation */

        long        s_lnnoptr;          /* file ptr to line number */

        unsigned short  s_nreloc;       /* number of relocation entries */

        unsigned short  s_nlnno;        /* number of line number entries */

        long        s_flags;            /* flags */

};

#define  SCNHDR  struct scnhdr
#define  SCNHSZ  sizeof(SCNHDR)
```

Figure 11-10. Section Header Declaration

.bss Section Header

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are 0. The same is true of the STYP_NOLOAD and STYP_DSECT sections.

Sections

Figure 11-1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a four-byte boundary in the file.

Link editor SECTIONS directives (see Chapter 12) allow users to, among other things:

- describe how input sections are to be combined
- direct the placement of output sections
- rename output sections

If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if several object files, each with a **.text** section, are linked together the output object file contains a single **.text** section made up of the combined input **.text** sections.

Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Figure 11-11.

Bytes	Declaration	Name	Description
0-3	**long int**	**r_vaddr**	(Virtual) address of reference
4-7	**long int**	**r_symndx**	Symbol table index
8-9	**unsigned short**	**r_type**	Relocation type

Figure 11-11. Relocation Section Contents

The first four bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Figure 11-12.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_RELBYTE	017	Direct 8-bit reference to the symbol's virtual address.
R_RELWORD	020	Direct 16-bit reference to the symbol's virtual address.
R_RELLONG	021	Direct 32-bit reference to the symbol's virtual address.
R_PCRBYTE	022	A "PC-relative" 8-bit reference to the symbol's virtual address.
R_PCRWORD	023	A "PC-relative" 16-bit reference to the symbol's virtual address.
R_PCRLONG	024	A "PC-relative" 32-bit reference to the symbol's virtual address.

Figure 11-12. Relocation Types

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 11-13. This declaration may be found in the header file **reloc.h**.

```
struct reloc
{
    long              r_vaddr;     /* virtual address of reference */

    long              r_symndx;    /* index into symbol table */

    unsigned short    r_type;      /* relocation type */
};

#define RELOC     struct reloc

#define RELSZ     10
```

Figure 11-13. Relocation Entry Declaration

Line Numbers

When invoked with the **–g** option, the **cc** and **f77** commands cause an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like **sdb**. All line numbers in a section are grouped by function as shown in Figure 11-14.

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

Figure 11-14. Line Number Grouping

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the

text corresponding to the line numbers. The line number entries are relative to the beginning of the function, and appear in increasing order of address.

Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 11-15.

```
struct lineno
{
        union
        {
                  long    l_symndx;    /* symtbl index of func name */

                  long    l_paddr;     /* paddr of line number */
        } l_addr;
        unsigned short    l_lnno;      /* line number */

};

#define LINENO       struct lineno
#define LINESZ       6
```

Figure 11-15. Line Number Entry Declaration

Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 11-16.

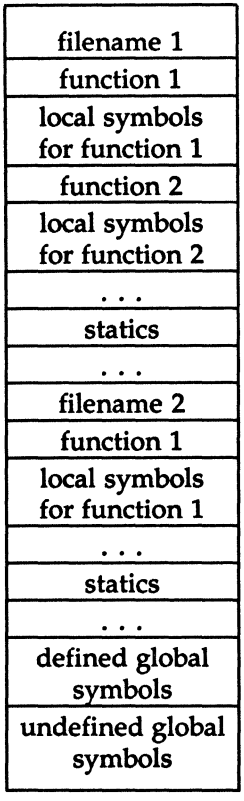


Figure 11-16. COFF Symbol Table

The word statics in Figure 11-16 means symbols defined with the C language storage class static outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

Special Symbols

The symbol table contains some special symbols that are generated by **as**. It contains other tools as well. These symbols are given in Figure 11-17.

Symbol	Meaning
.file	filename
.text	address of **.text** section
.data	address of **.data** section
.bss	address of **.bss** section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.*x*fake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
etext	next available address after the end of the output section **.text**
edata	next available address after the end of the output section **.data**
end	next available address after the end of the output section **.bss**

Figure 11-17. Special Symbols in the Symbol Table

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function. An **.*x*fake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is **.***x***fake**, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are **.0fake**, **.1fake**, and **.2fake**. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

Inner Blocks

The C language defines a block as a compound statement that begins and ends with braces: { and }. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol, **.bb**, is put in the symbol table immediately before the first local symbol of that block. Also a special symbol, **.eb**, is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 11-18.

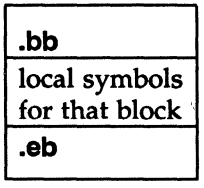


Figure 11-18. Special Symbols (**.bb** and **.eb**)

Because inner blocks can be nested by several levels, the **.bb**-**.eb** pairs and associated symbols may also be nested. See Figure 11-19.

```
{                                   /* block 1 */
        int i;
        char c;
        ...
        {                               /* block 2 */
                long a;
                ...
            {                               /* block 3 */
                        int x;
                        ....
            }                               /* block 3 */

        }                               /* block 2 */

        {                               /* block 4 */
                        long i;
                        ...
        }                               /* block 4 */
}                                   /* block 1 */
```

Figure 11-19. Nested blocks

The symbol table would look like Figure 11-20.

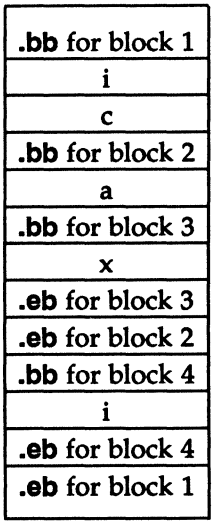


Figure 11-20. Example of the Symbol Table

Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 11-21.

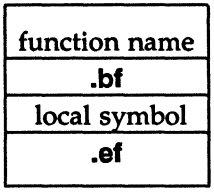


Figure 11-21. Symbols for Functions

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 11-22. Note that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Bytes	Declaration	Name	Description
0-7	(see text below)	**_n**	These 8 bytes contain either a symbol name or an index to a symbol
8-11	**long int**	**n_value**	Symbol value; storage class dependent
12-13	**short**	**n_scnum**	Section number of symbol
14-15	**unsigned short**	**n_type**	Basic and derived type specification
16	**char**	**n_sclass**	Storage class of symbol
17	**char**	**n_numaux**	Number of auxiliary entries

Figure 11-22. Symbol Table Entry Format

Symbol Names

The first eight bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the eight bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first four bytes serve to distinguish a symbol table entry with an offset from one with a name in the first eight bytes as shown in Figure 11-23.

Bytes	Declaration	Name	Description
0-7	**char**	**n_name**	8-character null-padded symbol name
0-3	**long**	**n_zeroes**	Zero in this field indicates the name is in the string table
4-7	**long**	**n_offset**	Offset of the name in the string table

Figure 11-23. Name Field

Special symbols generated by the C Compilation System are discussed above under "Special Symbols."

Storage Classes

The storage class field has one of the values described in Figure 11-24. These **#define**'s may be found in the header file **storclass.h**.

Mnemonic	Value	Storage Class
C_EFCN	–1	physical end of a function
C_NULL	0	–
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialized static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	filename
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

Figure 11-24. Storage Classes

All these storage classes except for C_ALIAS and C_HIDDEN are generated by the **cc** or **as** commands. The compress utility, **cprs**, generates the C_ALIAS mnemonic. This utility (described in the *User's Reference Manual*) removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class C_HIDDEN is not used by any operating system tools.

Some of these storage classes are used only internally by the C Compilation Systems. These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are given in Figure 11-25.

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.*x*fake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Figure 11-25. Storage Class by Special Symbols

Also some storage classes are used only for certain special symbols. They are summarized in Figure 11-26.

Storage Class	Special Symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

Figure 11-26. Restricted Storage Classes

Symbol Value Field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Figure 11-27.

Storage Class	Meaning of Value
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	(see text below)
C_ALIAS	tag index
C_HIDDEN	relocatable address

Figure 11-27. Storage Class and Value

If a symbol has storage class C_FILE, the value of that symbol equals the symbol table entry index of the next **.file** symbol. That is, the **.file** entries form a one-way linked list in the symbol table. If there are no more **.file** entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers are listed in Figure 11-28.

Mnemonic	Section Number	Meaning
N_DEBUG	–2	Special symbolic debugging symbol
N_ABS	–1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

Figure 11-28. Section Number

A special section number (–2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of –1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and **.eos** symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply defined external symbol (i.e., FORTRAN common or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0 and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the **.bss** section. The maximum size of all the input symbols with the same name is used to allocate

space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

Section Numbers and Storage Classes

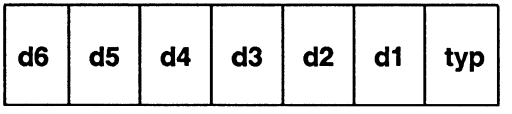
Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Figure 11-29.

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Figure 11-29. Section Number and Storage Class

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C Compilation System only if the **–g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is:



Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Figure 11-30.

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_VOID	1	void
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Figure 11-30. Fundamental Types

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Figure 11-31.

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

Figure 11-31. Derived Types

The following examples demonstrate the interpretation of the symbol table entry representing type.

```
char *func();
```

Here **func** is the name of a function that returns a pointer to a character. The fundamental type of **func** is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for **func** contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character.

```
short *tabptr[10][25][3];
```

Here **tabptr** is a three-dimensional array of pointers to short integers. The fundamental type of **tabptr** is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

Type Entries and Storage Classes

Figure 11-32 shows the type entries that are legal for each storage class.

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Figure 11-32. Type Entries by Storage Class

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in Figure 11-33. This declaration may be found in the header file **syms.h**.

```
struct syment
{
   union
   {
          char              _n_name[SYMNMLEN];    /* symbol name*/
          struct
          {
                   long     _n_zeroes;     /* symbol name */

                   long     _n_offset;     /* location in string table */
          } _n_n;
          char              *_n_nptr[2];   /* allows overlaying */
   } _n;
   unsigned long     n_value;                /* value of symbol */

   short             n_scnum;                /* section number */

   unsigned short    n_type;                 /* type and derived */

   char              n_sclass;               /* storage class */

   char              n_numaux;               /* number of aux entries */
};

#define  n_name          _n._n_name
#define  n_zeroes        _n._n_n._n_zeroes
#define  n_offset        _n._n_n._n_offset
#define  n_nptr          _n._n_nptr[1]

#define  SYMNMLEN  8
#define  SYMESZ    18    /* size of a symbol table entry */
```

Figure 11-33. Symbol Table Entry Declaration

11

Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Figure 11-34.

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	filename
.text,.data, .bss	C_STAT	DT_NON	T_NULL	section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
fcname	C_EXT C_STAT	DT_FCN	(Note 1)	function
arrname	(Note 2)	DT_ARY	(Note 1)	array
.bb,.eb	C_BLOCK	DT_NON	T_NULL	beginning and end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Notes to Figure 11-34:
1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

Figure 11-34. Auxiliary Symbol Table Entries

In Figure 11-34, *tagname* means any symbol name including the special symbol **.xfake**, and *fcname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Figure 11-34 should have a union format in its auxiliary entry.

NOTE

It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available, and should be obtained from the **n_numaux** field in the symbol table.

Filenames

Each of the auxiliary table entries for a filename contains a 14-character filename in bytes 0 through 13. The remaining bytes are 0.

Sections

The auxiliary table entries for sections have the format as shown in Figure 11-35.

Bytes	Declaration	Name	Description
0-3	**long int**	**x_scnlen**	section length
4-5	**unsigned short**	**x_nreloc**	number of relocation entries
6-7	**unsigned short**	**x_nlinno**	number of line numbers
8-17	–	–	unused (filled with zeroes)

Figure 11-35. Format for Auxiliary Table Entries for Sections

Tag Names

The auxiliary table entries for tag names have the format shown in Figure 11-36.

Bytes	Declaration	Name	Description
0-5	–	–	unused (filled with zeroes)
6-7	**unsigned short**	**x_size**	size of structure, union, and enumeration
8-11	–	–	unused (filled with zeroes)
12-15	**long int**	**x_endndx**	index of next entry beyond this structure, union, or enumeration
16-17	–	–	unused (filled with zeroes)

Figure 11-36. Tag Names Table Entries

End of Structures

The auxiliary table entries for the end of structures have the format shown in Figure 11-37.

Bytes	Declaration	Name	Description
0-3	**long int**	**x_tagndx**	tag index
4-5	–	–	unused (filled with zeroes)
6-7	**unsigned short**	**x_size**	size of structure, union, or enumeration
8-17	–	–	unused (filled with zeroes)

Figure 11-37. Table Entries for End of Structures

Functions

The auxiliary table entries for functions have the format shown in Figure 11-38.

Bytes	Declaration	Name	Description
0-3	**long int**	**x_tagndx**	tag index
4-7	**long int**	**x_fsize**	size of function (in bytes)
8-11	**long int**	**x_lnnoptr**	file pointer to line number
12-15	**long int**	**x_endndx**	index of next entry beyond this point
16-17	**unsigned short**	**x_tvndx**	index of the function's address in the transfer vector table (not used by the operating system)

Figure 11-38. Table Entries for Functions

Arrays

The auxiliary table entries for arrays have the format shown in Figure 11-39. Defining arrays having more than four dimensions produces a warning message.

Bytes	Declaration	Name	Description
0-3	**long int**	**x_tagndx**	tag index
4-5	**unsigned short**	**x_lnno**	line number of declaration
6-7	**unsigned short**	**x_size**	size of array
8-9	**unsigned short**	**x_dimen[0]**	first dimension
10-11	**unsigned short**	**x_dimen[1]**	second dimension
12-13	**unsigned short**	**x_dimen[2]**	third dimension
14-15	**unsigned short**	**x_dimen[3]**	fourth dimension
16-17	–	–	unused (filled with zeroes)

Figure 11-39. Table Entries for Arrays

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in Figure 11-40.

Bytes	Declaration	Name	Description
0-3	–	–	unused (filled with zeroes)
4-5	**unsigned short**	**x_lnno**	C-source line number
6-17	–	–	unused (filled with zeroes)

Figure 11-40. End of Block and Function Entries

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Figure 11-41.

Bytes	Declaration	Name	Description
0-3	–	–	unused (filled with zeroes)
4-5	**unsigned short**	**x_lnno**	C-source line number
6-11	–	–	unused (filled with zeroes)
12-15	**long int**	**x_endndx**	index of next entry past this block
16-17	–	–	unused (filled with zeroes)

Figure 11-41. Format for Beginning of Block and Function

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Figure 11-42.

Bytes	Declaration	Name	Description
0-3	**long int**	**x_tagndx**	tag index
4-5	–	–	unused (filled with zeroes)
6-7	**unsigned short**	**x_size**	size of the structure, union, or enumeration
8-17	–	–	unused (filled with zeroes)

Figure 11-42. Entries for Structures, Unions, and Enumerations

Aggregates defined by **typedef** may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;

struct people
{
        char name[20];
        long id;
};

typedef struct people EMPLOYEE;
```

The symbol EMPLOYEE has an auxiliary table entry in the symbol table but symbol STUDENT will not because it is a forward reference to a structure.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Figure 11-43. This declaration may be found in the header file **syms.h**.

```
union auxent
{
      struct
      {
            long   x_tagndx;
            union
            {
                struct
                {
                      unsigned short   x_lnno;
                      unsigned short   x_size;
                } x_lnsz;
                long    x_fsize;
            } x_misc;
            union
            {
                struct
 .
 .
 .
 .
 .
 .
                {
                      long    x_lnnoptr;
                      long    x_endndx;
                } x_fcn;
                struct
                {
                      unsigned short   x_dimen[DIMNUM];
                } x_ary;
            } x_fcnary;
            unsigned short   x_tvndx;
        } x_sym;
```

Figure 11-43. Auxiliary Symbol Table Entry (Sheet 1 of 2)

```
        struct
        {
            char    x_fname[FILNMLEN];
        } x_file;
        struct
        {
            long    x_scnlen;
            unsigned short    x_nreloc;
            unsigned short    x_nlinno;
        } x_scn;
        struct
        {
            long    x_tvfill;
            unsigned short    x_tvlen;
            unsigned short    x_tvran[2];
        } x_tv;
}
#define FILNMLEN   14
#define DIMNUM     4
#define AUXENT     union auxent
#define AUXESZ     18
```

Figure 11-43. Auxiliary Symbol Table Entry (Sheet 2 of 2)

String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer then eight characters, **long_name_1** and **another_one**) the string table has the format as shown in Figure 11-44.

'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

Figure 11-44. String Table

The index of **long_name_1** in the string table is 4 and the index of **another_one** is 16.

Access Routines

Operating system releases contain a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. a function that returns the symbol table index for a particular symbol

These routines can be found in the library **libld.a** and are listed in Section 3 of the *Programmer's Reference Manual.* A summary of what is available can be found in the *Programmer's Reference Manual* under **ldfcn**(4).