Appendix A

Common Object File Format

The TMS320C3x/C4x assembler and linker create object files that are in common object file format (COFF). COFF is an implementation of an object file format of the same name that was developed by AT&T for use on UNIX-based systems. This object file format has been chosen because it encourages modular programming and provides more powerful and flexible methods for managing code segments and target system memory.

One of the basic COFF concepts is *sections.* Chapter 2, *Introduction to Common Object File Format*, discusses COFF sections in detail. If you understand section operation, you will be able to use the TMS320 assembly language tools more efficiently.

This appendix contains technical details about COFF object file structure. Most of this information pertains to the symbolic debugging information that is produced by the TMS320C3x/C4x C compiler. The main purpose of this appendix is to provide supplementary information for those of you who are interested in the internal format of COFF object files.

Topics in this appendix include:

Topic **Page**

A.1 How the COFF File Is Structured

The elements of a COFF object file describe the file's sections and symbolic debugging information. These elements include:

- A file header
- Optional header information
- A table of section headers
- Raw data for each initialized section
- Relocation information for each initialized section
- Line number entries for each initialized section
- A symbol table
- A string table

The assembler and linker produce object files with the same COFF structure; however, a program that is linked for the final time will not contain relocation entries. Figure A–1 illustrates the overall object file structure.

Figure A–1. COFF File Structure

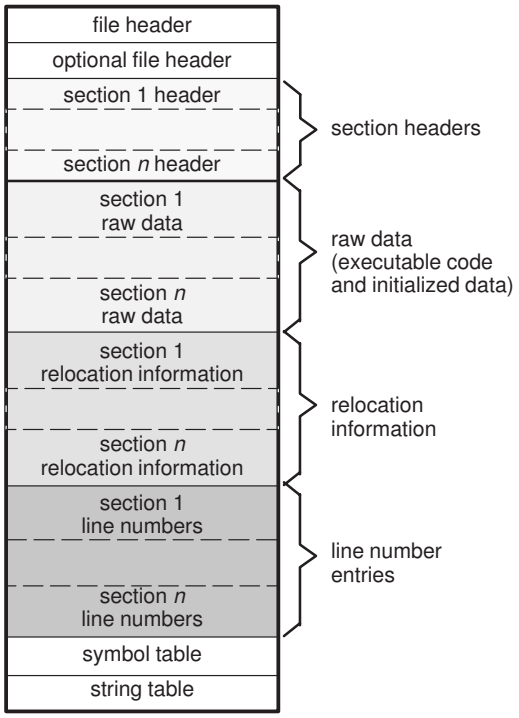
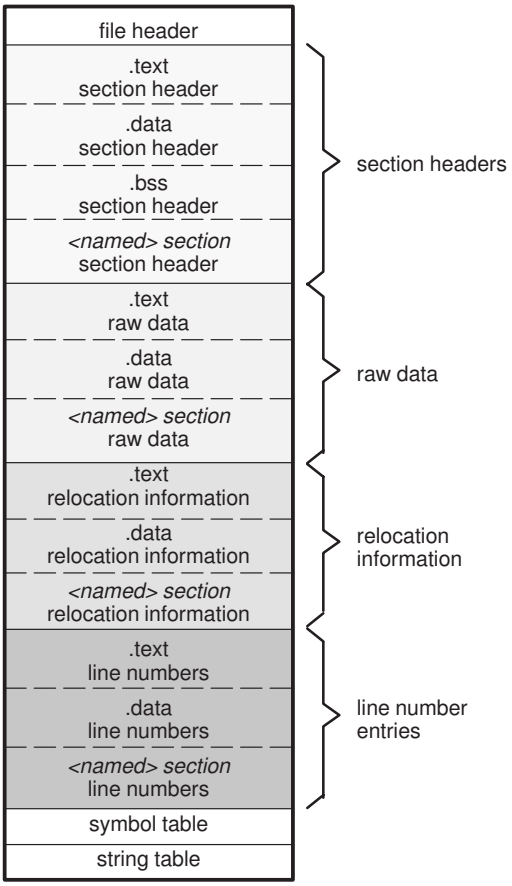


Figure A–2 shows a typical example of a COFF object file that contains the three default sections, .text, .data, and .bss, and a named section (referred to as <named>). By default, the tools place sections into the object file in the following order: .text, .data, initialized named sections, .bss, and uninitialized named sections. Although uninitialized sections have a section headers, they have no raw data, relocation information, or line number entries. This is because the .bss and .usect directives simply reserve space for uninitialized data; uninitialized sections contain no actual code.

Figure A–2. Sample COFF Object File



A.1.1 Impact of Switching Operating Systems

The TMS320C3x/C4x COFF files are recognized by all operating system versions of the development tools. When you switch from one operating system to another, only the file header information in the COFF files needs to be byte swapped. The raw data in the COFF files does not need any changes.

The development tools can detect the difference in the file headers and automatically compensate for it. This is true if using only TMS320C3x/C4x development tools.

To tell the difference between COFF files, you can look at the magic number in the optional file header. Bytes 0 and 1 contain the magic number. For the SunOS™ or HP-UX™ operating systems, the magic number is 108h. For the DOS operating system, the magic number is 801h.

A.2 File Header Structure

The file header contains 20 (COFF 0) or 22 (COFF 1 and 2) bytes of information that describe the general format of an object file. Table A–1 shows the structure of the file header.

Table A–1. File Header Contents

Byte Numbers	Type	Description
0–1	Unsigned short integer	COFF 0: Contains magic number (093h) to indicate the file can be executed in a TMS320C3x/C4x system. COFF 1 and 2: Contains COFF version number, either 0c1h (COFF1) or 0c2h (COFF 2)
2–3	Unsigned short integer	Number of section headers
4–7	Long integer	Time and date stamp; indicates when the file was created
8–11	Long integer	File pointer; contains the symbol table's starting address
12–15	Long integer	Number of entries in the symbol table
16–17	Unsigned short integer	Number of bytes in the optional header. This field is either 0 or 28; if it is 0, there is no optional file header.
18–19	Unsigned short integer	Flags (see Table A–2)
20–21	Unsigned short integer	Included for COFF 1 and 2 only. Contains magic number (093h) to indicate the file can be executed in a TMS320C3x/C4x system.

Table A–2 lists the flags that can appear in bytes 18 and 19 of the file header. Any number and combination of these flags can be set at the same time (for example, if bytes 18 and 19 are set to 0003h, F_RELFLG and F_EXEC are both set).

Table A–2. File Header Flags (Bytes 18 and 19)

Mnemonic	Flag	Description
F_VERS0	0000h	TMS320C30/C31 object code
F_RELFLG	0001h	Relocation information was stripped from the file
F_EXEC	0002h	The file is executable (it contains no unresolved external references)
F_LNNO	0004h	Line numbers were stripped from the file
F_LSYMS	0008h	Local symbols were stripped from the file
F_VERS1	0010h	TMS320C40/C44 object code
F_VERS2	0020h	TMS320C32 object code
F_LITTLE	0100h	Object data LSB first

A.3 Optional File Header Format

The linker creates the optional file header and uses it to perform relocation at download time. Partially linked files do not contain optional file headers. Table A–3 illustrates the optional file header format.

Table A–3. Optional File Header Contents

Byte Number	Type	Description
0–1	Short integer	Magic number (0108h)
2–3	Short integer	Version stamp
4–7	Long integer	Size (in words) of executable code
8–11	Long integer	Size (in words) of initialized data
12–15	Long integer	Size (in bits) of uninitialized data
16–19	Long integer	Entry point
20–23	Long integer	Beginning address of executable code
24–27	Long integer	Beginning address of initialized data

A.4 Section Header Structure

COFF object files contain a table of section headers that specify where each section begins in the object file. Each section has its own section header. The COFF0, COFF1, and COFF2 file types contain different section header information. Table A–4 shows the section header contents for COFF0 and COFF1 files. Table A–5 shows the section header contents for COFF2 files.

Table A–4. Section Header Contents for COFF 0 and COFF1 Files

Byte Number	Type	Description
0–7	Character	8-character section name, padded with nulls
8–11	Long integer	Section's physical address
12–15	Long integer	Section's virtual address
16–19	Long integer	Section size in words
20–23	Long integer	File pointer to raw data
24–27	Long integer	File pointer to relocation entries
28–31	Long integer	File pointer to line number entries
32–33	Unsigned short integer	Number of relocation entries
34–35	Unsigned short integer	Number of line number entries
36–37	Unsigned short integer	Flags (see Table A–6)
38	Character	Reserved
39	Unsigned character	Memory page number

Table A–5. Section Header Contents for COFF2 Files

Byte	Type	Description
0–7	Character	8-character section name, padded with nulls
8–11	Long integer	Section's physical address
12–15	Long integer	Section's virtual address
16–19	Long integer	Section size in words
20–23	Long integer	File pointer to raw data
24–27	Long integer	File pointer to relocation entries
28–31	Long integer	File pointer to line-number entries
32–35	Unsigned long	Number of relocation entries
36–39	Unsigned long	Number of line-number entries
40–43	Unsigned long	Flags (see Table A–6)
44–45	Short	Reserved
46–47	Unsigned short	Memory page number

Table A–6 lists the flags that can appear in the section header.

Table A–6. Section Header Flags

Mnemonic	Flag	Description
STYP_REG	0000h	Regular section (allocated, relocated, loaded)
STYP_DSECT	0001h	Dummy section (relocated, not allocated, not loaded)
STYP_NOLOAD	0002h	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0004h	Grouped section (formed from several input sections)
STYP_PAD	0008h	Padding section (loaded, not allocated, not relocated)
STYP_COPY	0010h	Copy section (relocated, loaded, but not allocated; relocation and line-number entries are processed normally)
STYP_TEXT	0020h	Section that contains executable code
STYP_DATA	0040h	Section that contains initialized data
STYP_BSS	0080h	Section that contains uninitialized data
STYP_ALIGN	0700h	Section that is aligned on a page boundary

Note: The term *loaded* means that the raw data for this section appears in the object file.

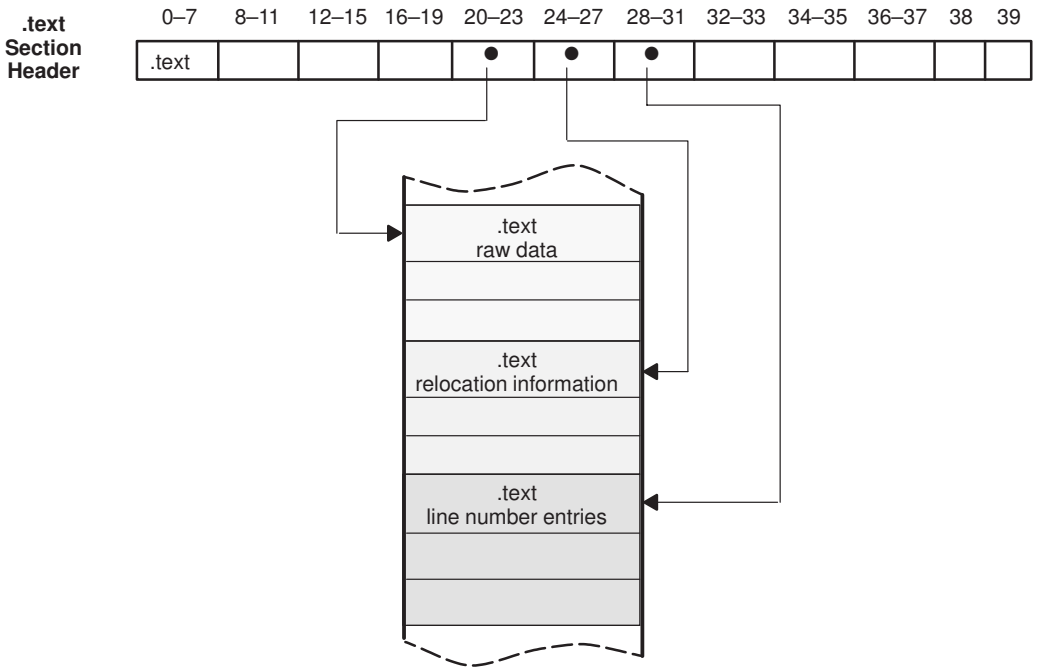
The flags listed in Table A–6 can be combined; for example, if the flags word is set to 024h, then both STYP_GROUP and STYP_TEXT are set. Bits 8–11 of the section header flags specify alignment. The alignment is 2^n where *n* is the value in bits 8–11.

The flags are in:

Bytes	For This COFF Format
36 and 37	COFF1
40 to 43	COFF2

Figure A–3 illustrates how the pointers in a section header would point to the various elements in an object file that are associated with the .text section.

Figure A–3. An Example of Section Header Pointers for the .text Section



As Figure A–2 on page A-3, shows, uninitialized sections (created with .bss and .usect directives) vary from this format. Although uninitialized sections have section headers, they have no raw data, relocation information, and line number information. They occupy no actual space in the object file. Therefore, the number of relocation entries, the number of line number entries, and the file pointers are 0 for an uninitialized section. The header of an uninitialized section simply tells the linker how much space for variables it should reserve in the memory map.

A.5 Structuring Relocation Information

A COFF object file has one relocation entry for each relocatable reference. The assembler automatically generates relocation entries. The linker reads the relocation entries as it reads each input section and performs relocation. The relocation entries determine how references within an input section are treated.

The relocation information entries use the format shown in Table A–7.

Table A–7. Relocation Entry Contents

Bytes	Type	Description
COFF 0:		
0–3	Long integer	Virtual address of the reference
4–5	Unsigned short integer	Symbol table index (0–65535)
6–7	Unsigned short integer	16 LSBs of reference, for R_PARTMS8
8–9	Unsigned short integer	Relocation type (see Table A–8)
COFF 1 and COFF 2:		
0–3	Long integer	Virtual address of the reference
4–7	Unsigned long integer	Symbol table index (0–65535)
8–9	Unsigned short integer	COFF 1: Reserved COFF 2: Additional byte used for extended address calculations
10–11	Unsigned short integer	Relocation type (see Table A–8)

The **virtual address** is the symbol's address in the current section *before* relocation; it specifies *where* a relocation must occur. (This is the address of the field in the object code that must be patched.)

Following is an example of code that generates a relocation entry:

```
0002                        .global   X
0003   0000   FF80          B         X
       0001   0000!
```

In this example, the virtual address of the relocatable field is 0001.

The **symbol table index** is the index of the referenced symbol. In the preceding example, this field would contain the index of X in the symbol table. The amount of the relocation is the difference between the symbol's current address in the section and its assembly-time address. The relocatable field must be relocated by the same amount as the referenced symbol. In the example, X has a value of 0 before relocation. Suppose X is relocated to address 2000h. This is the relocation amount (2000h – 0 = 2000h), so the relocation field at address 1 is patched by adding 2000h to it.

You can determine a symbol's relocated address if you know which section it is defined in. For example, if X is defined in .data and .data is relocated by 2000h, X is relocated by 2000h.

If the symbol table index in a relocation entry is –1 (0FFFFh), this is called an *internal relocation.* In this case, the relocation amount is simply the amount by which the current section is being relocated.

The **relocation type** specifies the size of the field to be patched and describes how the patched value should be calculated. The type field depends on the addressing mode that was used to generate the relocatable reference. In the preceding example, the actual address of the referenced symbol (X) will be placed in a 16-bit field in the object code. This is a 16-bit direct relocation, so the relocation type is R_RELWORD. Table A–8 lists the relocation types.

Table A–8. Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
R_ABS	0	Absolute address, no relocation
R_REL24	05	24-bit direct reference to symbol's address
R_RELWORD	020	16-bit direct reference to symbol's address
R_RELLONG	021	32-bit direct reference to symbol's address
R_PCRWORD	023	16-bit PC-relative reference
R_PCR24	025	24-bit PC-relative reference
R_PARTLS16	040	16-bit offset of 24-bit address
R_PARTMS8	041	8-bit page of 24-bit address (16-bit data page reference)

A.6 Line-Number Table Structure

The object file contains a table of line number entries that are useful for symbolic debugging. When the C compiler produces several lines of assembly language code, it creates a line number entry that maps these lines back to the original line of C source code that generated them. Each single line number entry contains 6 bytes of information. Table A–9 shows the format of a line-number entry.

Table A–9. Line-Number Entry Format

Byte Number	Type	Description
0–3	Long integer	This entry may have one of two values: 1) If it is the first entry in a block of line-number entries, it points to a symbol entry in the symbol table. 2) If it is not the first entry in a block, it is the physical address of the line indicated by bytes 4–5.
4–5	Unsigned short integer	This entry may have one of two values: 1) If this field is 0, this is the first line of a function entry. 2) If this field is *not* 0, this is the line number of a line of C source code.

Figure A–4 shows how line number entries are grouped into blocks.

Figure A–4. Line-Number Blocks

Symbol Index 1	**0**
physical address	line number
physical address	line number
Symbol Index *n*	**0**
physical address	line number
physical address	line number

As Figure A–4 shows, each entry is divided into halves:

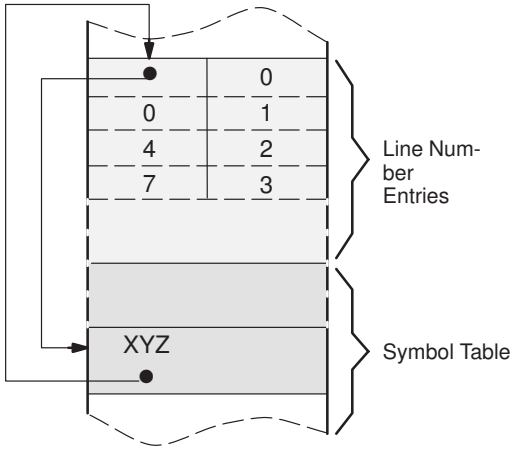
- For the *first line* of a function, bytes 0–3 point to the name of a symbol or a function in the symbol table and bytes 4–5 contain a 0, which indicates the beginning of a block.

- For the *remaining lines* in a function, bytes 0–3 show the physical address (the number of words created by a line of C source) and bytes 4–5 show the address of the original C source, relative to its appearance in the C source program.

The line entry table can contain many of these blocks.

Figure A–5 illustrates the line number entries for a function named XYZ. As shown, the function name is entered as a symbol in the symbol table. The first portion on XYZ's block of line number entries points to the function name in the symbol table. Assume that the original function in the C source contained three lines of code. The first line of code produces 4 words of assembly language code, the second line produces 3 words, and the third line produces 10 words.

Figure A–5. Line-Number Entries Example



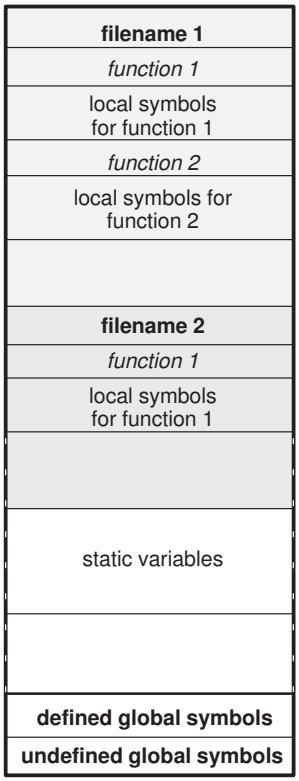
(Note that the symbol table entry for XYZ has a field that points back to the beginning of the line number block.)

Because line numbers are not often needed, the linker provides an option (-s) that strips line number information from the object file; this provides a more compact object module.

A.7 Symbol Table Structure and Content

The order of symbols in the symbol table is very important; they appear in the sequence shown in Figure A–6.

Figure A–6. Symbol Table Contents



Static variables refer to symbols defined in C that have storage class static outside any function. If you have several modules that use symbols with the same name, making them static confines the scope of each symbol to the module that defines it (this eliminates multiple-definition conflicts).

The entry for each symbol in the symbol table contains the symbol's:

- Name (or a pointer into the string table)
- Type
- Value
- Section it was defined in
- Storage class
- Basic type (integer, character, etc.)
- Derived type (array, structure, etc.)
- Dimensions
- Line number of the source code that defined the symbol

Section names are also defined in the symbol table.

All symbol entries, regardless of class and type, have the same format in the symbol table. Each symbol table entry contains the 18 bytes of information listed in Table A–10. Each symbol may also have an 18-byte auxiliary entry; the special symbols listed in Table A–11 on page A-16, always have an auxiliary entry. Some symbols may not have all the characteristics listed above; if a particular field is not set, it is set to null.

Table A–10. Symbol Table Entry Contents

Byte Number	Type	Description
0–7	Character	This field contains one of the following: 1) An 8-character symbol name, padded with nulls 2) An offset into the string table if the symbol name is longer than 8 characters
8–11	Long integer	Symbol value; storage class dependent
12–13	Short integer	Section number of the symbol
14–15	Unsigned short integer	Basic and derived type specification
16	Character	Storage class of the symbol
17	Character	Number of auxiliary entries (always 0 or 1)

A.7.1 Special Symbols

The symbol table contains some special symbols that are generated by the compiler, assembler, and linker. Each special symbol contains ordinary symbol table information and an auxiliary entry. Table A–11 lists these symbols.

Table A–11. Special Symbols in the Symbol Table

Symbol	Description
.file	File name
.text	Address of the .text section
.data	Address of the .data section
.bss	Address of the .bss section
.bb	Address of the beginning of a block
.eb	Address of the end of a block
.bf	Address of the beginning of a function
.ef	Address of the end of a function
.target	Pointer to a structure or union that is returned by a function
.*n*fake	Dummy tag name for a structure, union, or enumeration
.eos	End of a structure, union, or enumeration
etext	Next available address after the end of the .text output section
edata	Next available address after the end of the .data output section
end	Next available address after the end of the .bss output section

Several of these symbols appear in pairs:

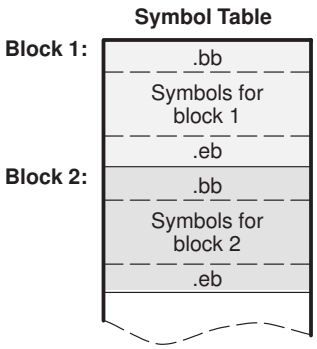
- .bb/.eb indicate the beginning and end of a block.
- .bf/.ef indicate the beginning and end of a function.
- *n*fake/.eos name and define the limits of structures, unions, and enumerations that were not named. The .eos symbol is also paired with named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler assigns it a name so that it can be entered into the symbol table. These names are of the form *n*fake, where *n* is an integer. The compiler begins numbering these symbol names at 0.

Symbols and Blocks

In C, a block is a compound statement that begins and ends with braces. A block always contains symbols. The symbol definitions for any particular block are grouped together in the symbol table, and are delineated by the .bb/.eb special symbols. Blocks can be nested in C, and their symbol table entries can be nested correspondingly. Figure A–7 shows how block symbols are grouped in the symbol table.

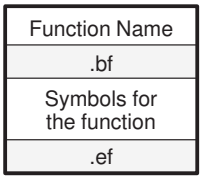
Figure A–7. Symbols for Blocks



Symbols and Functions

The symbol definitions for a function appear in the symbol table as a group, delineated by .bf/.ef special symbols. The symbol table entry for the function name precedes the .bf special symbol. Figure A–8 shows the format of symbol table entries for a function.

Figure A–8. Symbols for Functions



If a function returns a structure or union, then a symbol table entry for the special symbol .target will appear between the entries for the function name and the .bf special symbol.

A.7.2 Symbol Name Format

The first 8 bytes of a symbol table entry (bytes 0–7) indicate a symbol's name:

- If the symbol name is 8 characters or less, this field has type *character.* The name is padded with nulls (if necessary) and stored in bytes 0–7.
- If the symbol name is greater than 8 characters, this field is treated as two long integers. The entire symbol name is stored in the string table. Bytes 0–3 contain 0, and bytes 4–7 are an offset into the string table.

A.7.3 String Table Structure

Symbol names that are longer than eight characters are stored in the string table. The field in the symbol table entry that would normally contain the symbol's name contains, instead, a pointer to the symbol's name in the string table. Names are stored contiguously in the string table, delimited by a null byte. The first four bytes of the string table contain the size of the string table in bytes; thus, offsets into the string table are greater than or equal to four.

Figure A–9 is a string table that contains two symbol names, Adaptive–Filter and Fourier–Transform. The index in the string table is 4 for Adaptive–Filter and 20 for Fourier–Transform.

Figure A–9. Sample String Table

40			
'A'	'd'	'a'	'p'
't'	'i'	'v'	'e'
'—'	'F'	'i'	'l'
't'	'e'	'r'	'\0'
'F'	'o'	'u'	'r'
'i'	'e'	'r'	'—'
'T'	'r'	'a'	'n'
's'	'f'	'o'	'r'
'm'	'\0'	\0'	\0'

A.7.4 Storage Classes

Byte 16 of the symbol table entry indicates the storage class of the symbol. Storage classes refer to the method in which the C compiler accesses a symbol. Table A–12 lists valid storage classes.

Table A–12. Symbol Storage Classes

Mnemonic	Value	Storage Class	Mnemonic	Value	Storage Class
C_NULL	0	No storage class	C_USTATIC	14	Uninitialized static
C_AUTO	1	Automatic variable	C_ENTAG	15	Enumeration tag
C_EXT	2	External symbol	C_MOE	16	Member of an enumeration
C_STAT	3	Static	C_REGPARM	17	Register parameter
C_REG	4	Register variable	C_FIELD	18	Bit field
C_EXTDEF	5	External definition	C_UEXT	19	Tentative definition
C_LABEL	6	Label	C_STATLAB	20	Static .label symbol
C_ULABEL	7	Undefined label	C_EXTLAB	21	External .label symbol
C_MOS	8	Member of a structure	C_BLOCK	100	Beginning or end of a block; used only for the .bb and .eb special symbols
C_ARG	9	Function argument	C_FCN	101	Beginning or end of a function; used only for the .bf and .ef special symbols
C_STRTAG	10	Structure tag	C_EOS	102	End of structure; used only for the .eos special symbol
C_MOU	11	Member of a union	C_FILE	103	Filename; used only for the .file special symbol
C_UNTAG	12	Union tag	C_LINE	104	Used only by utility programs
C_TPDEF	13	Type definition			

Some special symbols are restricted to certain storage classes. Table A–13 lists these symbols and their storage classes.

Table A–13. Special Symbols and Their Storage Classes

Special Symbol	Restricted to This Storage Class	Special Symbol	Restricted to This Storage Class
.file	C_FILE	.eos	C_EOS
.bb	C_BLOCK	.text	C_STAT
.eb	C_BLOCK	.data	C_STAT
.bf	C_FCN	.bss	C_STAT
.ef	C_FCN		

A.7.5 Symbol Values

Bytes 8–11 of a symbol table entry indicate a symbol's value. A symbol's value depends on the symbol's storage class; Table A–14 summarizes the storage classes and related values.

Table A–14. Symbol Values and Storage Classes

Storage Class	Value Description	Storage Class	Value Description
C_AUTO	Stack offset in bits	C_UNTAG	0
C_EXT	Relocatable address	C_TPDEF	0
C_UEXT	0	C_NULL	none
C_EXTDEF	Relocatable address	C_USTATIC	0
C_STAT	Relocatable address	C_ENTAG	0
C_REG	Register number	C_MOE	Enumeration value
C_LABEL	Relocatable address	C_REGPARM	Register number
C_ULABLE	Relocatable address	C_STATLAB	Relocatable address
C_MOS	Offset in bits	C_FIELD	Bit displacement
C_ARG	Stack offset in bits	C_BLOCK	Relocatable address
C_STRTAG	0	C_FCN	Relocatable address
C_EOS	0	C_LINE	0
C_MOU	Offset in bits	C_FILE	0
C_EXTLAB	Relocatable address		

If a symbol's storage class is C_FILE, the symbol's value is a pointer to the next .file symbol. Thus, the .file symbols form a one-way linked list in the symbol table. When there are no more .file symbols, the final .file symbol points back to the first .file symbol in the symbol table.

The value of a relocatable symbol is its virtual address. When the linker relocates a section, the value of a relocatable symbol changes accordingly.

A.7.6 Section Number

Bytes 12–13 of a symbol table entry contain a number that indicates which section the symbol was defined in. Table A–15 lists these numbers and the sections they indicate.

Table A–15. Section Numbers

Mnemonic	Section Number	Description
N_DEBUG	–2	Special symbolic debugging symbol
N_ABS	–1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1	.text section (typical)
N_SCNUM	2	.data section (typical)
N_SCNUM	3	.bss section (typical)
N_SCNUM	4–32,767	Section number of a named section, in the order in which the named sections are encountered

If there were no .text, .data, or .bss sections, the numbering of named sections would begin with 1.

If a symbol has a section number of 0, –1, or –2, then it is not defined in a section. A section number of –2 indicates a symbolic debugging symbol, which includes structure, union, and enumeration tag names; type definitions; and filenames. A section number of –1 indicates that the symbol has a value but is not relocatable. A section number of 0 indicates a relocatable external symbol that is not defined in the current file.

A.7.7 Type Entry

Bytes 14–15 of the symbol table entry define the symbol's type. Each symbol has one basic type and one to six derived types.

Following is the format for this 16-bit type entry:

	Derived Type 6	Derived Type 5	Derived Type 4	Derived Type 3	Derived Type 2	Derived Type 1	Basic Type
Size (in bits):	2	2	2	2	2	2	4

Bits 0–3 of the type field indicate the basic type. Table A–16 lists valid basic types.

Table A–16. Basic Types

Mnemonic	Value	Type
T_NULL	0	Type not assigned
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 an enumeration
T_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short integer

Bits 4–15 of the type field are arranged as six 2-bit fields, which can indicate 1 to 6 derived types. Table A–17 lists the possible derived types.

Table A–17. Derived Types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARY	3	Array

An example of a symbol with several derived types would be a symbol with a type entry of 0000000011010011_2. This entry indicates that the symbol is a pointer to an array of short integers.

A.7.8 Auxiliary Entries

Each symbol table entry may have **one** or **no** auxiliary entry. An auxiliary symbol table entry contains the same number of bytes as a symbol table entry (18), but the format of an auxiliary entry depends on the symbol's type and storage class. Table A–18 summarizes these relationships.

Table A–18. Auxiliary Symbol Table Entries Format

Name	Storage Class	Type Entry		Auxiliary Entry Format
		Derived Type 1	Basic Type	
.file	C_FILE	DT_NON	T_NULL	Filename (see Table A–19)
.text, .data, .bss	C_STAT	DT_NON	T_NULL	Section (see Table A–20)
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	Tag name (see Table A–21)
.eos	C_EOS	DT_NON	T_NULL	End of structure (see Table A–22)
fcname	C_EXT C_STAT	DT_FCN	(See note 1)	Function (see Table A–23)
arrname	(See note 2)	DT_ARY	(See note 1)	Array (see Table A–24)
.bb, .eb	C_BLOCK	DT_NON	T_NULL	Beginning and end of a block (see Table A–25 and Table A–26)
.bf, .ef	C_FCN	DT_NON	T_NULL	Beginning and end of a function (see Table A–25 and Table A–26)
Name related to a structure, union, or enumeration	(See note 2)	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION T_ENUM	Name related to a structure, union, or enumeration (see Table A–27)

Notes: 1) Any except T_MOE
2) C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF

In Table A–18, *tagname* refers to any symbol name (including the special symbol *n*fake). *Fcname* and *arrname* refer to any symbol name.

A symbol that satisfies more than one condition in Table A–18 should have a union format in its auxiliary entry. A symbol that satisfies none of these conditions should not have an auxiliary entry.

Filenames

Each of the auxiliary table entries for a filename contains a 14-character file name in bytes 0–13. Bytes 14–17 are not used.

Table A–19. Filename Format for Auxiliary Table Entries

Byte Number	Type	Description
0–13	Character	File name
14–17	—	Unused

Sections

Table A–20 illustrates the format of the auxiliary table entries.

Table A–20. Section Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Section length
4–6	Unsigned short integer	Number of relocation entries
7–8	Unsigned short integer	Number of line number entries
9–17	—	Unused (zero filled)

Tag Names

Table A–21 illustrates the format of auxiliary table entries for tag names.

Table A–21. Tag Name Format for Auxiliary Table Entries

Byte Number	Type	Description
0–5	—	Unused (zero filled)
6–7	Unsigned short integer	Size of structure, union, or enumeration
8–11	—	Unused (zero filled)
12–15	Long integer	Index of next entry beyond this structure, union, or enumeration
16–17	—	Unused (zero filled)

End of Structure

Table A–22 illustrates the format of auxiliary table entries for ends of structures.

Table A–22. End-of-Structure Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	—	Unused (zero filled)
6–7	Unsigned short integer	Size of structure, union, or enumeration
8–17	—	Unused (zero filled)

Functions

Table A–23 illustrates the format of auxiliary table entries for functions.

Table A–23. Function Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–7	Long integer	Size of function (in bits)
8–11	Long integer	File pointer to line number
12–15	Long integer	Index of next entry beyond this function
16–17	—	Unused (zero filled)

Arrays

Table A–24 illustrates the format of auxiliary table entries for arrays.

Table A–24. Array Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	Unsigned short integer	Line number declaration
6–7	Unsigned short integer	Size of array
8–9	Unsigned short integer	First dimension
10–11	Unsigned short integer	Second dimension
12–13	Unsigned short integer	Third dimension
14–15	Unsigned short integer	Fourth dimension
16–17	—	Unused (zero filled)

End of Blocks and Functions

Table A–25 illustrates the format of auxiliary table entries for the ends of blocks and functions.

Table A–25. End-of-Blocks and Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	—	Unused (zero filled)
4–5	Unsigned short integer	C source line number
6–17	—	Unused (zero filled)

Beginning of Blocks and Functions

Table A–26 illustrates the format of auxiliary table entries for the beginnings of blocks and functions.

Table A–26. Beginning-of-Blocks and Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	—	Unused (zero filled)
4–5	Unsigned short integer	C source line number of block begin
6–11	—	Unused (zero filled)
12–15	Long integer	Index of next entry past this block
16–17	—	Unused (zero filled)

Names Related to Structures, Unions, and Enumerations

Table A–27 illustrates the format of auxiliary table entries for the names of structures, unions, and enumerations.

Table A–27. Structure, Union, and Enumeration Names Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	—	Unused (zero filled)
6–7	Unsigned short integer	Size of the structure, union, or enumeration
8–17	—	Unused (zero filled)