Application Report

SPRAAO8–April 2009

Common Object File Format

ABSTRACT

The assembler and link step create object files 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 format encourages modular programming and provides powerful and flexible methods for managing code segments and target system memory.

This appendix contains technical details about the Texas Instruments COFF object file structure. Much of this information pertains to the symbolic debugging information that is produced by the C compiler. The purpose of this application note is to provide supplementary information on the internal format of COFF object files.

Topic **Page**

1 COFF File Structure

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
- A symbol table
- A string table

The assembler and link step produce object files with the same COFF structure; however, a program that is linked for the final time does not usually contain relocation entries. Figure 1 illustrates the object file structure.

Figure 1. COFF File Structure

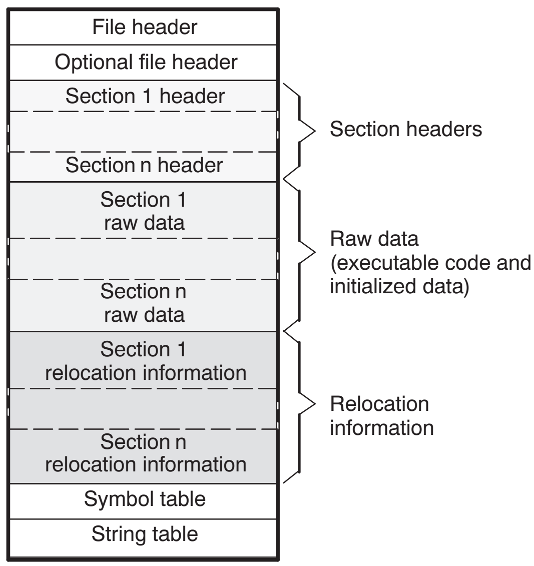
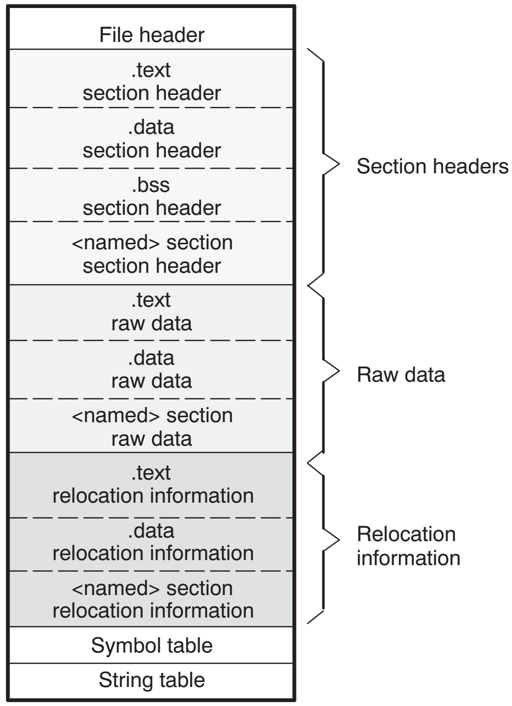


Figure 2 shows a typical example of a COFF object file that contains the three default sections, .text, .data, and .bss, and a named section (referred to as <named>). By default, the tools place sections into the object file in the following order: .text, .data, initialized named sections, .bss, and uninitialized named sections. Although uninitialized sections have section headers, notice that 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 2. Sample COFF Object File



2 File Header Structure

The file header contains 22 bytes of information that describe the general format of an object file. Table 1 shows the structure of the COFF file header.

Table 1. File Header Contents

Byte Number	Type	Description
0-1	Unsigned short	Version ID; indicates version of COFF file structure
2-3	Unsigned short	Number of section headers
4-7	Integer	Time and date stamp; indicates when the file was created
8-11	Integer	File pointer; contains the symbol table's starting address
12-15	Integer	Number of entries in the symbol table
16-17	Unsigned short	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	Flags (see Table 2)
20-21	Unsigned short	Target ID; magic number (see Table 3) indicates the file can be executed in a specific TI system

Table 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.

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

Mnemonic	Flag	Description
F_RELFLG	0001h	Relocation information was stripped from the file
F_EXEC	0002h	The file is relocatable (it contains no unresolved external references)
F_LNNO [(1)]	0004h	For TMS430 and TMS470 only: Line numbers were stripped from the file. For other targets: Reserved
F_LSYMS	0008h	Local symbols were stripped from the file
F_LITTLE	0100h	The target is a little-endian device
F_BIG [(1)]	0200h	For C6000, MSP430, and TMS470 only: The target is a big-endian device. For other targets: Reserved
F_SYMMERGE [(1)]	1000h	For C2800, MSP430, and TMS470: Duplicate symbols were removed. For C6000: Reserved

(1) No mnemonic is defined when the flag value is reserved.

Table 3 lists the magic number for each Texas Instruments device family.

Table 3. Magic Number

Magic Number	Device Family
0097h	TMS470
0098h	TMS320C5400
0099h	TMS320C6000
009Ch	TMS320C5500
009Dh	TMS320C2800
00A0h	MSP430
00A1h	TMS320C5500+

3 Optional File Header Format

The link step creates the optional file header and uses it to perform relocation at download time. Partially linked files do not contain optional file headers. Table 4 illustrates the optional file header format.

Table 4. Optional File Header Contents

Byte Number	Type	Description
0-1	Short	Optional file header magic number (0108h)
2-3	Short	Version stamp
4-7	Long [(1)]	Size (in bytes) of executable code
8-11	Long [(1)]	Size (in bytes) of initialized data
12-15	Long [(1)]	Size (in bytes) of uninitialized data
16-19	Long [(1)]	Entry point
20-23	Long [(1)]	Beginning address of executable code
24-27	Long [(1)]	Beginning address of initialized data

(1) For C6000 the type is integer.

4 Section Header Structure

COFF object files contain a table of section headers that define where each section begins in the object file. Each section has its own section header. Table 5 shows the structure of each section header.

Table 5. Section Header Contents

Byte Number	Type	Description
0-7	Character	This field contains one of the following: 1) An 8-character section name padded with nulls. 2) A pointer into the string table if the symbol name is longer than eight characters.
8-11	Long [(1)]	Section's physical address
12-15	Long [(1)]	Section's virtual address
16-19	Long [(1)]	Section size in bytes (C6000, C55x, TMS470 and TMS430) or words (C2800, C5400)
20-23	Long [(1)]	File pointer to raw data
24-27	Long [(1)]	File pointer to relocation entries
28-31	Long [(1)]	Reserved
32-35	Unsigned long [(2)]	Number of relocation entries
36-39	Unsigned long [(2)]	For TMS470 and TMS430 only: Number of line number entries. For other devices: Reserved
40-43	Unsigned long [(2)]	Flags (see Table 7)
44-45	Unsigned short	Reserved
46-47	Unsigned short	Memory page number

(1) For C6000 the type is integer.
(2) For C6000 the type is unsigned integer.

For C5400 only, object files can be produced in either of two formats: COFF1 or COFF2. For all other device families all COFF object files are in the COFF2 format. The COFF1 and COFF2 file types contain different section header information. Table 6 shows the section header contents for COFF1 files. Table 5 shows the section header contents for COFF2 files.

Table 6. Section Header Contents for COFF1

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

Table 7 lists the flags that can appear in bytes 40 through 43 (36-37 for COFF1) of the section header.

Table 7. Section Header Flags

Mnemonic	Flag	Description [1]
STYP_REG	00000000h	Regular section (allocated, relocated, loaded)
STYP_DSECT	00000001h	Dummy section (relocated, not allocated, not loaded)
STYP_NOLOAD	00000002h	Noload section (allocated, relocated, not loaded)
STYP_GROUP[2]	00000004h	Grouped section (formed from several input sections). Other devices: Reserved
STYP_PAD[2]	00000008h	Padding section (loaded, not allocated, not relocated). Other devices: Reserved
STYP_COPY	00000010h	Copy section (relocated, loaded, but not allocated; relocation entries are processed normally)
STYP_TEXT	00000020h	Section contains executable code
STYP_DATA	00000040h	Section contains initialized data
STYP_BSS	00000080h	Section contains uninitialized data
STYP_BLOCK[3]	00001000h	Alignment used as a blocking factor.
STYP_PASS[3]	00002000h	Section should pass through unchanged.
STYP_CLINK	00004000h	Section requires conditional linking
STYP_VECTOR[4]	00008000h	Section contains vector table.
STYP_PADDED[4]	00010000h	section has been padded.

(1) The term *loaded* means that the raw data for this section appears in the object file. Only allocated sections are written to target memory.
(2) Applies to C2800, C5400, and C5500 only.
(3) Reserved for C2800, C5400, and C5500.
(4) Applies to C6000 only.

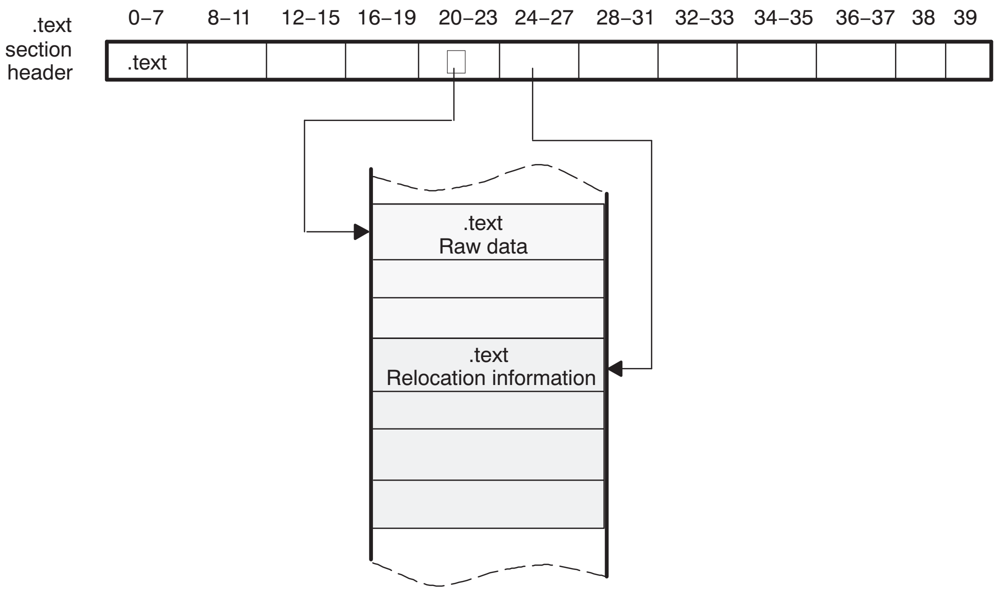
The flags listed in Table 7 can be combined; for example, if the flag's word is set to 060h, both STYP_DATA and STYP_TEXT are set.

Bits 8-11 of the section header flags are used for defining the alignment. The alignment is defined to be 2^(value of bits 8-11). For example if bits 8-11 are 0101b (decimal integer 5), then the alignment is 32 (2^5).

For MSP430 and TMS470, alignment is indicated by the bits masked by 0xF00. Alignment is the value in the bits raised to a power equal to the bit value. Alignment is 2 raised to the same power. For example, if the value in these 4 bits is 2, the alignment is 2 raised to the power 2 (or 4).

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

Figure 3. Section Header Pointers for the .text Section



As Figure 2 shows, uninitialized sections (created with the .bss and .usect directives) vary from this format. Although uninitialized sections have section headers, they have no raw data or relocation information; or, for MSP430 and TMS470, 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 link step how much space for variables it should reserve in the memory map.

5 Structuring Relocation Information

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

For C2800, C6000, MSP430, and TMS470, COFF file relocation information entries use the 10-byte format shown in Table 8.

Table 8. Relocation Entry Contents, 10-Byte Format

Byte Number	Type	Description
0-3	Long	Virtual address of the reference
4-5	Short	Symbol table index (0-65 535)
6-7	Unsigned short	Reserved
8-9	Unsigned short	Relocation type (see Table 11)

For C5400 and C5500, COFF file relocation information entries use the 12-byte format shown in Table 8.

Table 9. Relocation Entry Contents, 12-Byte Format

Byte Number	Type	Description
0-3	Long	Virtual address of the reference
4-7	Unsigned long	Symbol table index (0-65 535)
8-9	Unsigned short	For COFF1 files for C5400 only: Reserved For COFF2 files: Additional byte used for extended address calculations
10-11	Unsigned short	Relocation type (see Table 11)

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 C6000 code that generates a relocation entry:

```
2                         .global X
3 00000000 !00000012      b       X
```

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 contains 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 is calculated. The type field depends on the addressing mode that was used to generate the relocatable reference. In the preceding example for C6000, the actual address of the referenced symbol X is placed in an 8-bit field in the object code. This is an 8-bit address, so the relocation type is R_RELBYTE. The following tables list the relocation types by device family.

Table 10. Generic Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
RE_ADD	4000h	Addition (+)
RE_SUB	4001h	Subtraction (-)
RE_NEG	4002h	Negate (-)
RE_MPY	4003h	Multiplication (*)
RE_DIV	4004h	Division (/)
RE_MOD	4005h	Modulus (%)
RE_SR	4006h	Logical shift right (unsigned >>)
RE_ASR	4007h	Arithmetic shift right (signed >>)
RE_SL	4008h	Shift left (<<)
RE_AND	4009h	And (&)
RE_OR	400Ah	Or (|)
RE_XOR	400Bh	Exclusive Or (^)
RE_NOTB	400Ch	Not (~)
RE_ULDFLD	400Dh	Unsigned relocation field load
RE_SLDFLD	400Eh	Signed relocation field load

Table 10. Generic Relocation Types (Bytes 8 and 9) (continued)

Mnemonic	Flag	Relocation Type
RE_USTFLD	400Fh	Unsigned relocation field store
RE_SSTFLD	4010h	Signed relocation field store
RE_PUSH	4011h	Push symbol on the stack
RE_PUSHSK	4012h	Push signed constant on the stack
RE_PUSHUK	4013h	Push unsigned constant on the stack
RE_PUSHPC	4014h	Push current section PC on the stack
RE_DUP	4015h	Duplicate top-of-stack and push a copy
RE_XSTFLD	4016h	Relocation field store, signedness is irrelevant
RE_PUSHSV	C011h	Push symbol: SEGVALUE flag is set

Table 11. C6000 Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_RELBYTE	000Fh	8-bit direct reference to symbol's address
R_RELWORD	0010h	16-bit direct reference to symbol's address
R_RELLONG	0011h	32-bit direct reference to symbol's address
R_C60BASE	0050h	Data page pointer-based offset
R_C60DIR15	0051h	Load or store long displacement
R_C60PCR21	0052h	21-bit packet, PC relative
R_C60PCR10	0053h	10-bit Packet PC Relative (BDEC, BPOS)
R_C60LO16	0054h	MVK instruction low half register
R_C60HI16	0055h	MVKH or MVKLH high half register
R_C60SECT	0056h	Section-based offset
R_C60S16	0057h	Signed 16-bit offset for MVK
R_C60PCR7	0070h	7-bit Packet PC Relative (ADDKPC)
R_C60PCR12	0071h	12-bit Packet PC Relative (BNOP)

Table 12. C2800 Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_RELBYTE	000Fh	8-bit direct reference to symbol's address
R_RELWORD	0010h	16-bit direct reference to symbol's address
R_RELLONG	0011h	32-bit direct reference to symbol's address
R_PARTLS7	0028h	7-bit offset of a 22-bit address
R_PARTLS6	005Dh	6-bit offset of a 22-bit address
R_PARTMID10	005Eh	Middle 10 bits of a 22-bit address
R_REL22	005Fh	22-bit direct reference to a symbol's address
R_PARTMS6	0060h	Upper 6 bits of an 22-bit address
R_PARTS16	0061h	Upper 16 bits of an 22-bit address
R_C28PCR16	0062h	PC relative 16-bit address
R_C28PCR8	0063h	PC relative 8-bit address
R_C28PTR	0064h	22-bit pointer
R_C28HI16	0065h	High 16 bits of address data
R_C28LOPTR	0066h	Pointer to low 64K
R_C28NWORD	0067h	16-bit negated relocation
R_C28NBYTE	0068h	8-bit negated relocation

Table 12. C2800 Relocation Types (Bytes 8 and 9) (continued)

Mnemonic	Flag	Relocation Type
R_C28HIBYTE	0069h	High 8 bits of a 16-bit data
R_C28RELS13	006Ah	Signed 13-bit value relocated as a 16-bit value

Table 13. C5400 Relocation Types (Bytes 10 and 11)

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_REL24	0005h	24-bit reference to symbol's address
R_RELBYTE	0017h	8-bit direct reference to symbol's address
R_RELWORD	0020h	16-bit direct reference to symbol's address
R_RELLONG	0021h	32-bit direct reference to symbol's address
R_PARTLS7	0028h	7 LSBs of an address
R_PARTMS9	0029h	9 MSBs of an address
R_REL13	002Ah	13-bit direct reference to symbol's address

Table 14. C5500 Relocation Types (Bytes 10 and 11)

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_REL24	0005h	24-bit direct reference to symbol's address
R_RELBYTE	0017h	8-bit direct reference to symbol's address
R_RELWORD	0020h	16-bit direct reference to symbol's address
R_RELLONG	0021h	32-bit direct reference to symbol's address
R_LD3_DMA	0170h	7 MSBs of a byte, unsigned; used in DMA address
R_LD3_MDP	0172h	7 bits spanning 2 bytes, unsigned; used as MDP register value
R_LD3_PDP	0173h	9 bits spanning 2 bytes, unsigned; used as PDP register value
R_LD3_REL23	0174h	23-bit unsigned value in 24-bit field
R_LD3_k8	0210h	8-bit unsigned direct reference
R_LD3_k16	0211h	16-bit unsigned direct reference
R_LD3_K8	0212h	8-bit signed direct reference
R_LD3_K16	0213h	16-bit signed direct reference
R_LD3_I8	0214h	8-bit unsigned PC-relative reference
R_LD3_I16	0215h	16-bit unsigned PC-relative reference
R_LD3_L8	0216h	8-bit signed PC-relative reference
R_LD3_L16	0217h	16-bit signed PC-relative reference
R_LD3_k4	0220h	Unsigned 4-bit shift immediate
R_LD3_k5	0221h	Unsigned 5-bit shift immediate
R_LD3_K5	0222h	Signed 5-bit shift immediate
R_LD3_k6	0223h	Unsigned 6-bit shift immediate
R_LD3_k12	0224h	Unigned 12-bit shift immediate

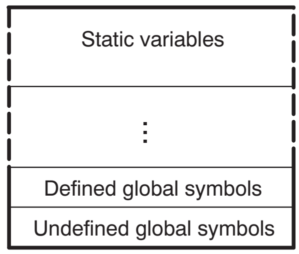
Table 15. MSP430 and TMS470 Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
R_RELLONG	0011h	32-bit direct reference to symbol's address
R_PCR23H	0016h	23-bit PC-relative reference to a symbol's address, in halfwords (divided by 2)
R_PCR24W	0017h	24-bit PC-relative reference to a symbol's address, in words (divided by 4)

6 Symbol Table Structure and Content

The order of symbols in the symbol table is very important; they appear in the sequence shown in Figure 4.

Figure 4. Symbol Table Contents



Static variables refer to symbols defined in C/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 an offset into the string table)
- Type
- Value
- Section it was defined in
- Storage class

For MSP430 and TMS470, the entry for each symbol in the symbol table also contains the symbol's:

- 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 16. Each symbol may also have an 18-byte auxiliary entry; the special symbols listed in Table 17 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 16. Symbol Table Entry Contents

Byte Number	Type	Description
0-7	Char	This field contains one of the following: 1) An 8-character symbol name, padded with nulls. 2) A pointer into the string table if the symbol name is longer than eight characters.
8-11	Long[(1)]	Symbol value; storage class dependent
12-13	Short	Section number of the symbol
14-15	Unsigned short	Reserved
16	Char	Storage class of the symbol
17	Char	Number of auxiliary entries (always 0 or 1)

(1) For C6000 the type is integer.

6.1 Special Symbols

The symbol table contains some special symbols that are generated by the compiler, assembler, and link step. Each special symbol contains ordinary symbol table information as well as an auxiliary entry. Table 17 lists these symbols.

Table 17. Special Symbols in the Symbol Table

Symbol	Description
.text	Address of the .text section
.data	Address of the .data section
.bss	Address of the .bss section
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

6.2 Symbol Name Format

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

- If the symbol name is eight 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 eight characters, this field is treated as two 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.

6.3 String Table Structure

The string table stores symbols with names longer than eight characters. The field in the symbol table entry that would normally contain the symbol's name actually points to the symbol's name in the string table. The string table contiguously stores names, delimited by a null byte. The first four bytes of the table contain the table size in bytes; thus, offsets into the string table are greater than or equal to 4.

Figure 5 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 5. String Table Entries for Sample Symbol Names

38 bytes

4 bytes			
'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'		

6.4 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/C++ compiler accesses a symbol. Table 18 lists valid storage classes.

Table 18. Symbol Storage Classes

Mnemonic	Value	Storage Class	Mnemonic	Value	Storage Class
C_NULL	0	No storage class	C_USTATIC	14	Undefined static
C_AUTO	1	Reserved	C_ENTAG	15	Reserved
C_EXT	2	External definition	C_MOE	16	Reserved
C_STAT	3	Static	C_REGPARM	17	Reserved
C_REG	4	Reserved	C_FIELD	18	Reserved
C_EXTREF	5	External reference	C_UEXT[1]	19	Tentative external definition
C_LABEL	6	Label	C_STATLAB[1]	20	Static load time label
C_ULABEL	7	Undefined label	C_EXTLAB[1]	21	External load time label
C_MOS	8	Reserved	C_VARARG[1][2]	27	Last declared parameter of a function with a variable number of arguments
C_ARG	9	Reserved	C_BLOCK	100	Reserved
C_STRTAG	10	Reserved	C_FCN	101	Reserved
C_MOU	11	Reserved	C_EOS	102	Reserved
C_UNTAG	12	Reserved	C_FILE	103	Reserved
C_TPDEF	13	Reserved	C_LINE	104	Used only by utility programs

(1) Not applicable to C5400 or C5500
(2) Not applicable to C2800

The .text, .data, and .bss symbols are restricted to the C_STAT storage class.

6.5 Symbol Values

Bytes 8-11 of a symbol table entry indicate a symbol's value. The C_EXT, C_STAT, and C_LABEL storage classes hold relocatable addresses.

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

6.6 Section Number

Bytes 12-13 of a symbol table entry contain a number that indicates in which section the symbol was defined. Table 19 lists these numbers and the indicated sections.

Table 19. Section Numbers

Mnemonic	Section Number	Description
None	-2	Reserved
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
None[1]	1	.text section (typical)
None[1]	2	.data section (typical)
None[1]	3	.bss section (typical)
None[1]	4-32 767	Section number of a named section, in the order in which the named sections are encountered

(1) For C5500 and C2800, the mnemonic is N_SCNUM

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, it is not defined in a section. 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.

6.7 Auxiliary Entries

Each symbol table entry can have *one* or *no*auxiliary entry. An auxiliary symbol table entry contains the same number of bytes as a symbol table entry (18). Table 20 illustrates the format of auxiliary table entries.

Table 20. Section Format for Auxiliary Table Entries

Byte Number	Type	Description
0-3	Long[1]	Section length
4-5	Unsigned short	Number of relocation entries
6-7	Unsigned short	Number of line number entries
8-17		Not used (zero filled)

(1) For C6000 the type is integer.

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265