

*Preliminary*

# ***TMS320C6000 Assembly Language Tools v 6.0 Beta***

## ***User's Guide***

Literature Number: SPRU186P  
October 2006



*Preliminary*

<b>Preface</b>	<b>13</b>
<b>1 Introduction to the Software Development Tools</b>	<b>17</b>
1.1 Software Development Tools Overview	18
1.2 Tools Descriptions	19
<b>2 Introduction to Common Object File Format</b>	<b>21</b>
2.1 Sections	22
2.2 How the Assembler Handles Sections	23
2.2.1 Uninitialized Sections	23
2.2.2 Initialized Sections	24
2.2.3 Named Sections	24
2.2.4 Subsections	25
2.2.5 Section Program Counters	26
2.2.6 Using Sections Directives	26
2.3 How the Linker Handles Sections	29
2.3.1 Default Memory Allocation	29
2.3.2 Placing Sections in the Memory Map	30
2.4 Relocation	30
2.5 Run-Time Relocation	32
2.6 Loading a Program	32
2.7 Symbols in a COFF File	33
2.7.1 External Symbols	33
2.7.2 The Symbol Table	33
<b>3 Assembler Description</b>	<b>35</b>
3.1 Assembler Overview	36
3.2 The Assembler's Role in the Software Development Flow	36
3.3 Invoking the Assembler	38
3.4 Naming Alternate Directories for Assembler Input	40
3.4.1 Using the -I Assembler Option	40
3.4.2 Using the C6X_A_DIR or A_DIR Environment Variable	41
3.5 Source Statement Format	42
3.5.1 Label Field	43
3.5.2 Mnemonic Field	43
3.5.3 Unit Specifier Field	44
3.5.4 Operand Field	44
3.5.5 Comment Field	44
3.6 Constants	45
3.6.1 Binary Integers	45
3.6.2 Octal Integers	45
3.6.3 Decimal Integers	45
3.6.4 Hexadecimal Integers	46
3.6.5 Character Constants	46
3.6.6 Assembly-Time Constants	46
3.7 Character Strings	47

3.8	Symbols .....	47
3.8.1	Labels .....	47
3.8.2	Local Labels.....	47
3.8.3	Symbolic Constants .....	50
3.8.4	Defining Symbolic Constants (-ad Option) .....	50
3.8.5	Predefined Symbolic Constants .....	51
3.8.6	Substitution Symbols.....	53
3.9	Expressions .....	54
3.9.1	Operators .....	54
3.9.2	Expression Overflow and Underflow .....	54
3.9.3	Well-Defined Expressions .....	55
3.9.4	Conditional Expressions .....	55
3.9.5	Legal Expressions.....	55
3.9.6	Expression Examples.....	55
3.10	Source Listings .....	57
3.11	Debugging Assembly Source .....	58
3.12	Cross-Reference Listings .....	60
<b>4</b>	<b>Assembler Directives .....</b>	<b>61</b>
4.1	Directives Summary .....	62
4.2	Directives That Define Sections .....	65
4.3	Directives That Initialize Constants .....	67
4.4	Directives That Perform Alignment and Reserve Space .....	68
4.5	Directives That Format the Output Listings .....	69
4.6	Directives That Reference Other Files .....	70
4.7	Directives That Enable Conditional Assembly.....	71
4.8	Directives That Define Unions or Structures.....	72
4.9	Directives That Define Symbols at Assembly Time.....	74
4.10	Miscellaneous Directives .....	75
4.11	Directives Reference.....	76
<b>5</b>	<b>Macro Language .....</b>	<b>127</b>
5.1	Using Macros.....	128
5.2	Defining Macros .....	128
5.3	Macro Parameters/Substitution Symbols .....	129
5.3.1	Directives That Define Substitution Symbols.....	130
5.3.2	Built-In Substitution Symbol Functions.....	132
5.3.3	Recursive Substitution Symbols .....	133
5.3.4	Forced Substitution .....	133
5.3.5	Accessing Individual Characters of Subscripted Substitution Symbols.....	134
5.3.6	Substitution Symbols as Local Variables in Macros .....	135
5.4	Macro Libraries.....	136
5.5	Using Conditional Assembly in Macros .....	137
5.6	Using Labels in Macros .....	139
5.7	Producing Messages in Macros .....	140
5.8	Using Directives to Format the Output Listing .....	141
5.9	Using Recursive and Nested Macros .....	141
5.10	Macro Directives Summary .....	142
<b>6</b>	<b>Archiver Description .....</b>	<b>145</b>
6.1	Archiver Overview .....	146

6.2	The Archiver's Role in the Software Development Flow.....	147
6.3	Invoking the Archiver .....	148
6.4	Archiver Examples.....	149
<b>7</b>	<b>Linker Description .....</b>	<b>151</b>
7.1	Linker Overview .....	152
7.2	The Linker's Role in the Software Development Flow .....	153
7.3	Invoking the Linker.....	154
7.4	Linker Options .....	155
7.4.1	Relocation Capabilities (-a and -r Options) .....	156
7.4.2	Create an Absolute Listing File (-abs Option) .....	156
7.4.3	Allocate Memory for Use by the Loader to Pass Arguments (--args Option) .....	157
7.4.4	Disable Merge of Symbolic Debugging Information (-b Option) .....	157
7.4.5	C Language Options (-c and -cr Options) .....	157
7.4.6	Define an Entry Point (-e global_symbol Option) .....	157
7.4.7	Set Default Fill Value (-f fill_value Option).....	158
7.4.8	Make a Symbol Global (-g symbol Option) .....	158
7.4.9	Make All Global Symbols Static (-h Option).....	158
7.4.10	Define Heap Size (-heap size Option) .....	159
7.4.11	Alter the Library Search Algorithm (-l Option, -L Option, and C_DIR/C6X_C_DIR Environment Variables) .....	159
7.4.12	Disable Conditional Linking (-j Option).....	160
7.4.13	Create a Map File (-m filename Option) .....	161
7.4.14	Name an Output Module (-o Option) .....	161
7.4.15	Strip Symbolic Information (-s Option) .....	162
7.4.16	Define Stack Size (-stack size Option).....	162
7.4.17	Generate Far Call Trampolines (--trampolines Option) .....	162
7.4.18	Introduce an Unresolved Symbol (-u symbol Option).....	164
7.4.19	Display a Message When an Undefined Output Section Is Created (-w Option) .....	164
7.4.20	Exhaustively Read and Search Libraries (-x and -priority Options) .....	165
7.4.21	Generate XML Link Information File (--xml_link_info Option).....	165
7.5	Linker Command Files .....	166
7.5.1	Reserved Names in Linker Command Files.....	167
7.5.2	Constants in Linker Command Files .....	167
7.6	Object Libraries .....	168
7.7	The MEMORY Directive .....	169
7.7.1	Default Memory Model .....	169
7.7.2	MEMORY Directive Syntax .....	169
7.8	The SECTIONS Directive .....	171
7.8.1	SECTIONS Directive Syntax .....	171
7.8.2	Allocation.....	173
7.8.3	Specifying Input Sections .....	177
7.8.4	Using Multi-Level Subsections .....	178
7.8.5	Allocation Using Multiple Memory Ranges .....	179
7.8.6	Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges .....	180
7.8.7	Allocating an Archive Member to an Output Section.....	181
7.9	Specifying a Section's Run-Time Address .....	182
7.9.1	Specifying Load and Run Addresses .....	182
7.9.2	Uninitialized Sections.....	183

7.9.3	Referring to the Load Address by Using the .label Directive.....	183
7.10	Using UNION and GROUP Statements .....	185
7.10.1	Overlaying Sections With the UNION Statement.....	185
7.10.2	Grouping Output Sections Together .....	186
7.10.3	Nesting UNIONS and GROUPs.....	187
7.10.4	Checking the Consistency of Allocators .....	187
7.11	Special Section Types (DSECT, COPY, and NOLOAD) .....	188
7.12	Default Allocation Algorithm .....	188
7.12.1	How the Allocation Algorithm Creates Output Sections .....	189
7.12.2	Reducing Memory Fragmentation .....	190
7.13	Assigning Symbols at Link Time.....	190
7.13.1	Syntax of Assignment Statements.....	190
7.13.2	Assigning the SPC to a Symbol .....	190
7.13.3	Assignment Expressions.....	191
7.13.4	Symbols Defined by the Linker .....	192
7.13.5	Assigning Exact Start, End, and Size Values of a Section to a Symbol.....	193
7.13.6	Why the Dot Operator Does Not Always Work .....	193
7.13.7	Address and Dimension Operators.....	194
7.14	Creating and Filling Holes .....	196
7.14.1	Initialized and Uninitialized Sections .....	196
7.14.2	Creating Holes .....	196
7.14.3	Filling Holes .....	197
7.14.4	Explicit Initialization of Uninitialized Sections .....	198
7.15	Linker-Generated Copy Tables .....	198
7.15.1	A Current Boot-Loaded Application Development Process .....	198
7.15.2	An Alternative Approach .....	199
7.15.3	Overlay Management Example .....	200
7.15.4	Generating Copy Tables Automatically With the Linker .....	200
7.15.5	The table() Operator.....	201
7.15.6	Boot-Time Copy Tables.....	202
7.15.7	Using the table() Operator to Manage Object Components.....	202
7.15.8	Copy Table Contents.....	202
7.15.9	General Purpose Copy Routine.....	204
7.15.10	Linker Generated Copy Table Sections and Symbols.....	204
7.15.11	Splitting Object Components and Overlay Management .....	205
7.16	Partial (Incremental) Linking.....	207
7.17	Linking C/C++ Code .....	208
7.17.1	Run-Time Initialization .....	208
7.17.2	Object Libraries and Run-Time Support .....	208
7.17.3	Setting the Size of the Stack and Heap Sections .....	208
7.17.4	Autoinitialization of Variables at Run Time .....	209
7.17.5	Initialization of Variables at Load Time .....	209
7.17.6	The -c and -cr Linker Options .....	210
7.18	Linker Example.....	210
<b>8</b>	<b>Absolute Lister Description .....</b>	<b>213</b>
8.1	Producing an Absolute Listing .....	214
8.2	Invoking the Absolute Lister .....	215
8.3	Absolute Lister Example .....	216

<b>9</b>	<b>Cross-Reference Lister Description</b>	<b>219</b>
9.1	Producing a Cross-Reference Listing	220
9.2	Invoking the Cross-Reference Lister	220
9.3	Cross-Reference Listing Example	221
<b>10</b>	<b>Object File Utilities Descriptions</b>	<b>223</b>
10.1	Invoking the Object File Display Utility	224
10.2	XML Tag Index	224
10.3	Example XML Consumer	228
10.3.1	The Main Application	228
10.3.2	xml.h Declaration of the XMLEntity Object	231
10.3.3	xml.cpp Definition of the XMLEntity Object	231
10.4	Invoking the Disassembler	234
10.5	Invoking the Name Utility	235
10.6	Invoking the Strip Utility	235
<b>11</b>	<b>Hex Conversion Utility Description</b>	<b>237</b>
11.1	The Hex Conversion Utility's Role in the Software Development Flow	238
11.2	Invoking the Hex Conversion Utility	239
11.2.1	Invoking the Hex Conversion Utility From the Command Line	239
11.2.2	Invoking the Hex Conversion Utility With a Command File	241
11.3	Understanding Memory Widths	242
11.3.1	Target Width	242
11.3.2	Specifying the Memory Width	243
11.3.3	Partitioning Data Into Output Files	244
11.3.4	Specifying Word Order for Output Words	246
11.4	The ROMS Directive	246
11.4.1	When to Use the ROMS Directive	247
11.4.2	An Example of the ROMS Directive	248
11.5	The SECTIONS Directive	250
11.6	Excluding a Specified Section	251
11.7	Assigning Output Filenames	252
11.8	Image Mode and the -fill Option	253
11.8.1	Generating a Memory Image	253
11.8.2	Specifying a Fill Value	253
11.8.3	Steps to Follow in Using Image Mode	253
11.9	Building a Table for an On-Chip Boot Loader	254
11.9.1	Description of the Boot Table	254
11.9.2	The Boot Table Format	254
11.9.3	How to Build the Boot Table	254
11.9.4	Using the C6x Boot Loader	256
11.10	Controlling the ROM Device Address	257
11.11	Description of the Object Formats	258
11.11.1	ASCII-Hex Object Format (-a Option)	258
11.11.2	Intel MCS-86 Object Format (-i Option)	259
11.11.3	Motorola Exorciser Object Format (-m Option)	260
11.11.4	Texas Instruments SDSMAC Object Format (-t Option)	260
11.11.5	Extended Tektronix Object Format (-x Option)	261
11.12	Hex Conversion Utility Error Messages	262
<b>12</b>	<b>Sharing C/C++ Header Files With Assembly Source</b>	<b>263</b>

12.1	Overview of the .cdecls Directive .....	264
12.2	Notes on C/C++ Conversions .....	264
12.2.1	Comments .....	264
12.2.2	Conditional Compilation (#if/#else/#ifdef/etc.).....	265
12.2.3	Pragmas .....	265
12.2.4	The #error and #warning Directives .....	265
12.2.5	Predefined symbol __ASM_HEADER__ .....	265
12.2.6	Usage Within C/C++ asm( ) Statements.....	265
12.2.7	The #include Directive .....	265
12.2.8	Conversion of #define Macros .....	265
12.2.9	The #undef Directive .....	266
12.2.10	Enumerations .....	266
12.2.11	C Strings.....	266
12.2.12	C/C++ Built-In Functions .....	267
12.2.13	Structures and Unions .....	267
12.2.14	Function/Variable Prototypes .....	267
12.2.15	C Constant Suffixes .....	268
12.2.16	Basic C/C++ Types .....	268
12.3	Notes on C++ Specific Conversions .....	268
12.3.1	Name Mangling .....	268
12.3.2	Derived Classes.....	268
12.3.3	Templates.....	269
12.3.4	Virtual Functions .....	269
12.4	New Assembler Support .....	269
12.4.1	Enumerations (.enum/.emember/.endenum).....	269
12.4.2	The .define Directive .....	269
12.4.3	The .undefine/.unasg Directives .....	270
12.4.4	The \$defined( ) Directive .....	270
12.4.5	The \$sizeof Built-In Function .....	270
12.4.6	Structure/Union Alignment & \$alignof( ) .....	270
12.4.7	The .cstring Directive .....	270
<b>A</b>	<b>Common Object File Format .....</b>	<b>271</b>
A.1	COFF File Structure .....	272
A.2	File Header Structure.....	273
A.3	Optional File Header Format .....	274
A.4	Section Header Structure .....	274
A.5	Structuring Relocation Information .....	276
A.6	Symbol Table Structure and Content .....	277
A.6.1	Special Symbols .....	278
A.6.2	Symbol Name Format .....	278
A.6.3	String Table Structure .....	278
A.6.4	Storage Classes .....	279
A.6.5	Symbol Values .....	279
A.6.6	Section Number .....	280
A.6.7	Auxiliary Entries .....	280
<b>B</b>	<b>Symbolic Debugging Directives .....</b>	<b>281</b>
B.1	DWARF Debugging Format .....	282
B.2	COFF Debugging Format.....	282



---

B.3	Debug Directive Syntax .....	283
<b>C</b>	<b>XML Link Information File Description .....</b>	<b>285</b>
C.1	XML Information File Element Types .....	286
C.2	Document Elements .....	286
C.2.1	Header Elements .....	286
C.2.2	Input File List .....	287
C.2.3	Object Component List.....	288
C.2.4	Logical Group List .....	289
C.2.5	Placement Map .....	291
C.2.6	Far Call Trampoline List .....	292
C.2.7	Symbol Table.....	293
<b>D</b>	<b>Glossary .....</b>	<b>295</b>

## List of Figures

1-1	TMS320C6000 Software Development Flow .....	18
2-1	Partitioning Memory Into Logical Blocks .....	22
2-3	Object Code Generated by the File in Figure 2-2.....	28
2-4	Combining Input Sections to Form an Executable Object Module.....	29
3-1	The Assembler in the TMS320C6000 Software Development Flow .....	37
3-2	Example Assembler Listing .....	58
4-1	The .field Directive .....	67
4-2	Initialization Directives .....	68
4-3	The .align Directive.....	68
4-4	The .space and .bes Directives .....	69
4-5	Double-Precision Floating-Point Format .....	91
4-6	The .field Directive .....	97
4-7	Single-Precision Floating-Point Format .....	97
4-8	The .usect Directive .....	125
6-1	The Archiver in the TMS320C6000 Software Development Flow.....	147
7-1	The Linker in the TMS320C6000 Software Development Flow .....	153
7-2	Section Allocation Defined by Example 7-4 .....	173
7-3	Run-Time Execution of Example 7-9.....	184
7-4	Memory Allocation Shown in Example 7-11 and Example 7-12.....	186
7-5	Autoinitialization at Run Time .....	209
7-6	Initialization at Load Time .....	210
8-1	Absolute Lister Development Flow .....	214
9-1	The Cross-Reference Lister in the TMS320C6000 Software Development Flow.....	220
11-1	The Hex Conversion Utility in the TMS320C6000 Software Development Flow .....	238
11-2	Hex Conversion Utility Process Flow.....	242
11-3	COFF Data and Memory Widths .....	243
11-4	Data, Memory, and ROM Widths .....	245
11-5	The infile.out File Partitioned Into Four Output Files.....	249
11-6	ASCII-Hex Object Format.....	258
11-7	Intel Hexadecimal Object Format.....	259
11-8	Motorola-S Format.....	260
11-9	TI-Tagged Object Format .....	261
11-10	Extended Tektronix Object Format .....	261
A-1	COFF File Structure .....	272
A-2	Sample COFF Object File .....	273
A-3	Section Header Pointers for the .text Section.....	275
A-4	Symbol Table Contents .....	277
A-5	String Table Entries for Sample Symbol Names .....	279

## List of Tables

3-1	CPU Control Registers .....	52
3-2	Processor Symbols .....	52
3-3	Assembler Version Symbols .....	53
3-4	Operators Used in Expressions (Precedence) .....	54
3-5	Symbol Attributes.....	60
4-1	Directives That Define Sections .....	62
4-2	Directives That Initialize Constants (Data and Memory) .....	62
4-3	Directives That Perform Alignment and Reserve Space .....	63
4-4	Directives That Format the Output Listing .....	63
4-5	Directives That Reference Other Files .....	63
4-6	Directives That Enable Conditional Assembly.....	63
4-7	Directives That Define Unions or Structures.....	64
4-8	Directives That Define Symbols at Assembly Time.....	64
4-9	Directives That Perform Miscellaneous Functions .....	65
5-1	Substitution Symbol Functions and Return Values.....	132
5-2	Creating Macros .....	143
5-3	Manipulating Substitution Symbols .....	143
5-4	Conditional Assembly .....	143
5-5	Producing Assembly-Time Messages.....	143
5-6	Formatting the Listing .....	143
7-1	Linker Options Summary .....	155
7-2	Groups of Operators Used in Expressions (Precedence) .....	192
9-1	Symbol Attributes in Cross-Reference Listing .....	222
10-1	XML Tag Index.....	224
11-1	Basic Hex Conversion Utility Options .....	240
11-2	Boot-Loader Options .....	255
11-3	Options for Specifying Hex Conversion Formats .....	258
A-1	File Header Contents.....	273
A-2	File Header Flags (Bytes 18 and 19).....	274
A-3	Optional File Header Contents.....	274
A-4	Section Header Contents .....	274
A-5	Section Header Flags (Bytes 40 Through 43) .....	275
A-6	Relocation Entry Contents.....	276
A-7	Relocation Types (Bytes 8 and 9) .....	276
A-8	Symbol Table Entry Contents .....	278
A-9	Special Symbols in the Symbol Table .....	278
A-10	Symbol Storage Classes.....	279
A-11	Section Numbers.....	280
A-12	Section Format for Auxiliary Table Entries .....	280
B-1	Symbolic Debugging Directives .....	283





## Read This First

---

### About This Manual

The *TMS320C6000 Assembly Language Tools User's Guide* tells you how to use these assembly language tools:

- Assembler
- Archiver
- Linker
- Absolute lister
- Cross-reference lister
- Object file display utility
- Disassembler
- Name utility
- Strip utility
- Hex conversion utility

Before you use this book, you should install the assembly language tools.

### How to Use This Manual

This book helps you learn how to use the Texas Instruments assembly language tools designed specifically for the TMS320C6000™ 32-bit devices. This book consists of four parts:

- **Introductory information**, consisting of [Chapter 1](#) and [Chapter 2](#), gives you an overview of the assembly language development tools. It also discusses common object file format (COFF), which helps you to use the TMS320C6000 tools more efficiently. Read [Chapter 2](#), *Introduction to Common Object File Format*, before using the assembler and linker.
- **Assembler description**, consisting of [Chapter 3](#) through [Chapter 5](#), contains detailed information about using the assembler. This portion explains how to invoke the assembler and discusses source statement format, valid constants and expressions, assembler output, and assembler directives. It also describes the macro language.
- **Additional assembly language tools**, consisting of [Chapter 6](#) through [Chapter 11](#), describes in detail each of the tools provided with the assembler to help you create executable object files. For example, [Chapter 7](#) explains how to invoke the linker, how the linker operates, and how to use linker directives; [Chapter 11](#) explains how to use the hex conversion utility.
- **Reference material**, consisting of [Appendix A](#) through [Appendix D](#), provides technical data about the internal format and structure of COFF object files. It discusses symbolic debugging directives that the TMS320C6000 C/C++ compiler uses. Finally, it includes hex conversion utility examples, assembler and linker error messages, and a glossary.

## Notational Conventions

This document uses the following conventions:

- The TMS320C62x™, C64x™, and C67x™ core is referred to as TMS320C6000 or C6000™.
- Program listings, program examples, and interactive displays are shown in a *special typeface*. Interactive displays use a bold version of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

Here is a sample program listing:

```

1 00000000          .data
2 00000000 0000002F  x      .byte  47
3 00000001 00000032  z      .byte  50
4 00000000          .text
5 00000000 010401E0      ADD    A0,A1,A2

```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered. Following is an example of command-line syntax.

```
cl6x -z [options] filename1 ... filenamen
```

The **cl6x -z** command invokes the linker and has two parameters. The first parameter, *options*, is optional (see the next bullet for details). The second parameter, *filename*, is required and you can enter more than one.

- Square brackets ( [ and ] ) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold typeface**, do not enter the brackets themselves. The following is an example of a command that has an optional parameter.

```
hex6x [options] filename
```

The **hex6x** command has two parameters. The second parameter, *filename*, is required. The first parameter, *options*, is optional. Since options is plural, you can select several options.

- In assembler syntax statements, column 1 is reserved for the first character of a label or symbol. If the label or symbol is optional, it is usually not shown. If it is a required parameter, it is shown starting against the left margin of the shaded box, as in the example below. No instruction, command, directive, or parameter other than a symbol or label can begin in column 1.

```
symbol .usect "section name", size in bytes [, alignment]
```

The *symbol* is required for the .usect directive and must begin in column 1. The *section name* must be enclosed in quotes and the parameter *size in bytes* must be separated from the *section name* by a comma. The *alignment* is optional and, if used, must be separated by a comma.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, each separated from the previous one by a comma.

```
.byte value1 [, ... , valuen]
```

- In program listings and program examples, pipe symbols (||) indicate parallel instructions, and square brackets ( [ ] ) indicate conditional instructions. In this example of parallel and conditional instructions, the instruction on line five executes in parallel with instruction on line six. The instruction on line eight is conditional: the branch to \$1 only occurs if the contents of A1 are not equal to 0:

```

1          .global tab1, tab2
2

```

```

3 00000000 00000028!      MVK      tab1,A0
4 00000004 00000068!      MVKH     tab1,A0
5 00000008 008031A9        MVK      99, A1
6 0000000c 010848C0 ||     ZERO     A2
7
8 00000010 80000212 $1:[A1] B      $1
9 00000014 01003674        STW      A2, *A0++
10 00000018 0087E1A0        SUB      A1,1,A1
11 0000001c 00004000        NOP      3

```

- Following are other symbols and abbreviations used throughout this document:

Symbol	Definition
B, b	Suffix — binary integer
H, h	Suffix — hexadecimal integer
LSB	Least significant bit
MSB	Most significant bit
0x	Prefix — hexadecimal integer
Q, q	Suffix — octal integer

## Related Documentation From Texas Instruments

The following books describe the TMS320C6000 devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number.

**[SPRU328](#) — *Code Composer Studio User's Guide*.** Explains how to use the Code Composer development environment to build and debug embedded real-time DSP applications.

**[SPRU187](#) — *TMS320C6000 Optimizing Compiler v6.0 Beta User's Guide*.** Describes the TMS320C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the TMS320C6000 platform of devices (including the C64x+ and C67x+ generations). The assembly optimizer helps you optimize your assembly code.

**[SPRU198](#) — *TMS320C6000 Programmer's Guide*.** Reference for programming the TMS320C6000 digital signal processors (DSPs). Before you use this manual, you should install your code generation and debugging tools. Includes a brief description of the C6000 DSP architecture and code development flow, includes C code examples and discusses optimization methods for the C code, describes the structure of assembly code and includes examples and discusses optimizations for the assembly code, and describes programming considerations for the C64x DSP.

**[SPRU189](#) — *TMS320C6000 DSP CPU and Instruction Set Reference Guide*.** Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C6000 digital signal processors (DSPs).

**[SPRU190](#) — *TMS320C6000 DSP Peripherals Overview Reference Guide*.** Provides an overview and briefly describes the peripherals available on the TMS320C6000 family of digital signal processors (DSPs).

**[SPRU197](#) — *TMS320C6000 Technical Brief*.** Provides an introduction to the TMS320C62x and TMS320C67x digital signal processors (DSPs) of the TMS320C6000 DSP family. Describes the CPU architecture, peripherals, development tools and third-party support for the C62x and C67x DSPs.

## **Trademarks**

TMS320C6000, TMS320C62x, C64x, C67x, C6000 are trademarks of Texas Instruments.

Intel is a trademark of Intel Corporation.

Windows is a trademark of Microsoft Corporation.

Windows is a registered trademark of Microsoft Corporation.

Motorola-S is a trademark of Motorola, Inc.

Tektronix is a trademark of Tektronix, Inc.

UNIX is a trademark of The Open Group.

UNIX is a registered trademark of licensed exclusively through X/Open Company Limited.



## ***Introduction to the Software Development Tools***

The TMS320C6000™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembly optimizer, an assembler, a linker, and assorted utilities. This chapter provides an overview of these tools.

The TMS320C6000 is supported by the following assembly language development tools:

- Assembler
- Archiver
- Linker
- Absolute lister
- Cross-reference lister
- Object file display utility
- Disassembler
- Name utility
- Strip utility
- Hex conversion utility

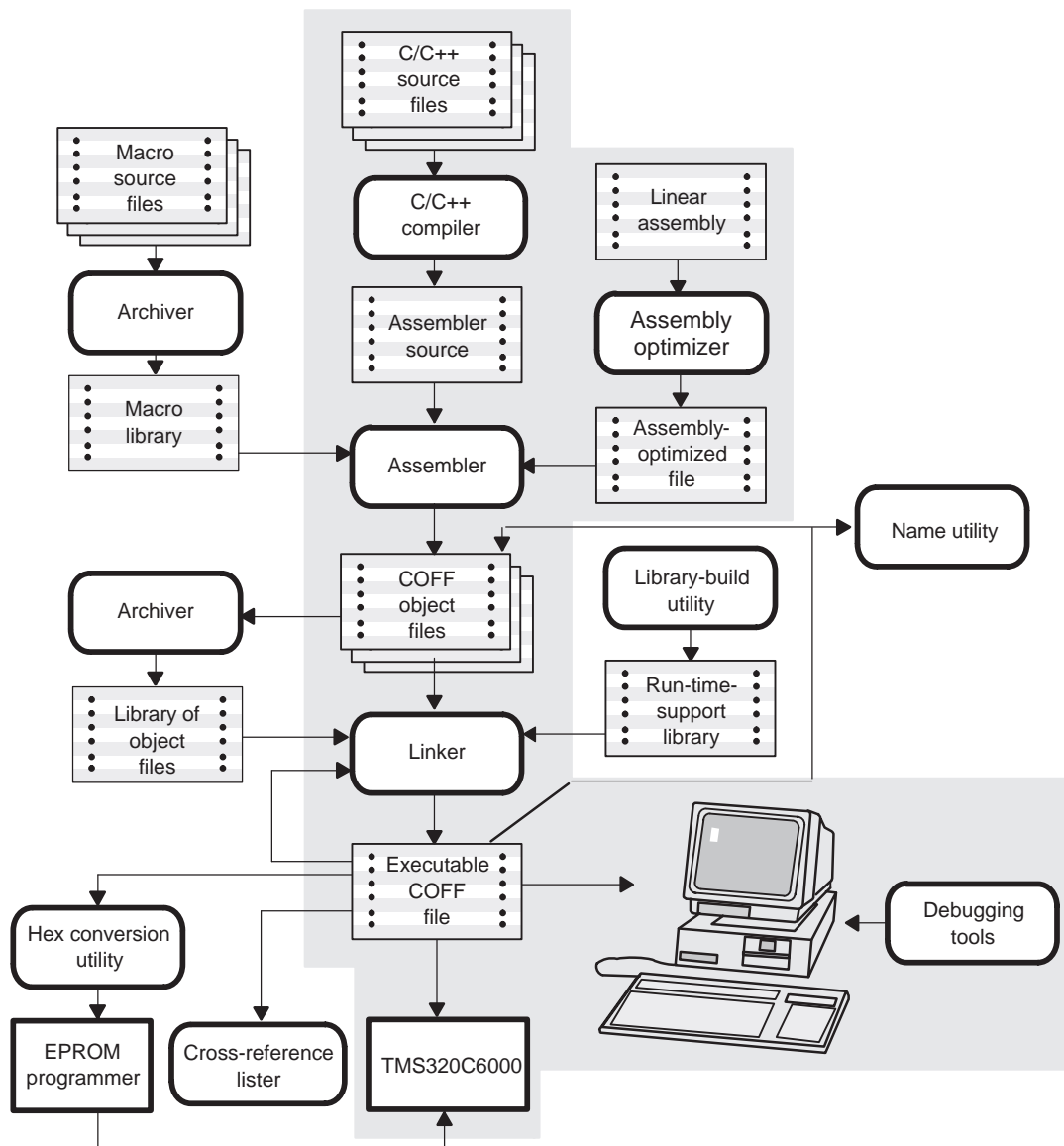
This chapter shows how these tools fit into the general software tools development flow and gives a brief description of each tool. For convenience, it also summarizes the C/C++ compiler and debugging tools. For detailed information on the compiler and debugger, and for complete descriptions of the TMS320C6000, refer to books listed in *Related Documentation From Texas Instruments*.

Topic	Page
<b>1.1 Software Development Tools Overview.....</b>	<b>18</b>
<b>1.2 Tools Descriptions .....</b>	<b>19</b>

## 1.1 Software Development Tools Overview

Figure 1-1 shows the TMS320C6000 software development flow. The shaded portion highlights the most common development path; the other portions are optional. The other portions are peripheral functions that enhance the development process.

**Figure 1-1. TMS320C6000 Software Development Flow**



## 1.2 Tools Descriptions

The following list describes the tools that are shown in [Figure 1-1](#):

- The **assembly optimizer** allows you to write linear assembly code without being concerned with the pipeline structure or with assigning registers. It assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly that takes advantage of software pipelining.  
See the *TMS320C6000 Optimizing Compiler User's Guide* for more information.
- The **C/C++ compiler** accepts C/C++ source code and produces TMS320C6000 assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:
  - The shell program enables you to compile, assemble, and link source modules in one step.
  - The optimizer modifies code to improve the efficiency of C/C++ programs.
  - The interlist utility interlists C/C++ source statements with assembly language output to correlate code produced by the compiler with your source code.
 See the *TMS320C6000 Optimizing Compiler User's Guide* for more information.
- The **assembler** translates assembly language source files into machine language COFF object files. Source files can contain instructions, assembler directives, and macro directives. You can use assembler directives to control various aspects of the assembly process, such as the source listing format, data alignment, and section content. See [Chapter 3, Assembler Description](#), through [Chapter 5, Macro Language](#), for more information. See the *TMS320C6000 CPU and Instruction Set Reference Guide* for detailed information on the assembly language instruction set.
- The **linker** combines object files into a single executable COFF object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files (created by the assembler) as input. It also accepts archiver library members and output modules created by a previous linker run. Linker directives allow you to combine object file sections, bind sections or symbols to addresses or within memory ranges, and define or redefine global symbols. See [Chapter 7, Linker Description](#), for more information.
- The **archiver** allows you to collect a group of files into a single archive file, called a library. For example, you can collect several macros into a macro library. The assembler searches the library and uses the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members. See [Chapter 6, Archiver Description](#), for more information.
- You can use the **library-build utility** to build your own customized run-time-support library. See the *TMS320C6000 Optimizing Compiler User's Guide* for more information.
- The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-Hex, Intel™, Motorola-S™, or Tektronix™ object format. The converted file can be downloaded to an EPROM programmer. See [Chapter 11, Hex Conversion Utility Description](#), for more information.
- The **absolute lister** uses linked object files to create .abs files. These files can be assembled to produce a listing of the absolute addresses of object code. See [Chapter 8, Absolute Lister Description](#), for more information.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definition, and their references in the linked source files. See [Chapter 9, Cross-Reference Lister Description](#), for more information.
- The main product of this development process is a module that can be executed in a **TMS320C6000** device. You can use one of several debugging tools to refine and correct your code. Available products include:
  - An instruction-accurate and clock-accurate software simulator
  - An XDS emulator

For information about these debugging tools, see the *Code Composer Studio User's Guide*.

In addition, the following utilities are provided:

- The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both human readable and XML formats. See [Section 10.1, Invoking the Object File Display Utility](#), for more information.

*Tools Descriptions*

---

- The **disassembler** writes the disassembled object code from object or executable files. See [Section 10.4, Invoking the Disassembler](#), for more information.
- The **name utility** prints a list of names defined and referenced in a COFF object or an executable file. See [Section 10.5, Invoking the Name Utility](#), for more information.
- The **strip utility** removes symbol table and debugging information from object and executable files. See [Section 10.6, Invoking the Strip Utility](#), for more information.

## Introduction to Common Object File Format

The assembler and linker create object files that can be executed by a TMS320C6000™ device. The format for these object files is called common object file format (COFF).

COFF makes modular programming easier because it encourages you to think in terms of *blocks* of code and data when you write an assembly language program. These blocks are known as sections. Both the assembler and the linker provide directives that allow you to create and manipulate sections.

This chapter focuses on the concept and use of sections in assembly language programs. See [Appendix A](#), *Common Object File Format*, for details about COFF object file structure.

Topic	Page
2.1 Sections.....	<a href="#">22</a>
2.2 How the Assembler Handles Sections .....	<a href="#">23</a>
2.3 How the Linker Handles Sections .....	<a href="#">29</a>
2.4 Relocation.....	<a href="#">30</a>
2.5 Run-Time Relocation .....	<a href="#">32</a>
2.6 Loading a Program.....	<a href="#">32</a>
2.7 Symbols in a COFF File .....	<a href="#">33</a>

## 2.1 Sections

The smallest unit of an object file is called a *section*. A section is a block of code or data that occupies contiguous space in the memory map with other sections. Each section of an object file is separate and distinct. COFF object files always contain three default sections:

<b>.text section</b>	usually contains executable code
<b>.data section</b>	usually contains initialized data
<b>.bss section</b>	usually reserves space for uninitialized variables

In addition, the assembler and linker allow you to create, name, and link *named* sections that are used like the .data, .text, and .bss sections.

There are two basic types of sections:

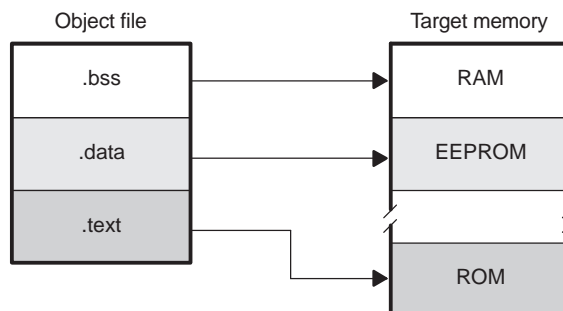
<b>Initialized sections</b>	contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.
<b>Uninitialized sections</b>	reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created with the .usect assembler directive are also uninitialized.

Several assembler directives allow you to associate various portions of code and data with the appropriate sections. The assembler builds these sections during the assembly process, creating an object file organized as shown in [Figure 2-1](#).

One of the linker's functions is to relocate sections into the target system's memory map; this function is called *allocation*. Because most systems contain several types of memory, using sections can help you use target memory more efficiently. All sections are independently relocatable; you can place any section into any allocated block of target memory. For example, you can define a section that contains an initialization routine and then allocate the routine into a portion of the memory map that contains ROM.

[Figure 2-1](#) shows the relationship between sections in an object file and a hypothetical target memory.

**Figure 2-1. Partitioning Memory Into Logical Blocks**



## 2.2 How the Assembler Handles Sections

The assembler identifies the portions of an assembly language program that belong in a given section. The assembler has five directives that support this function:

- .bss
- .usect
- .text
- .data
- .sect

The .bss and .usect directives create *uninitialized sections*; the .text, .data, and .sect directives create *initialized sections*.

You can create subsections of any section to give you tighter control of the memory map. Subsections are created using the .sect and .usect directives. Subsections are identified with the base section name and a subsection name separated by a colon. See [Section 2.2.4, Subsections](#), for more information.

### Default Sections Directive

**Note:** If you do not use any of the sections directives, the assembler assembles everything into the .text section.

### 2.2.1 Uninitialized Sections

Uninitialized sections reserve space in TMS320C6000 memory; they are usually allocated into RAM. These sections have no actual contents in the object file; they simply reserve memory. A program can use this space at run time for creating and storing variables.

Uninitialized data areas are built by using the .bss and .usect assembler directives.

- The .bss directive reserves space in the .bss section.
- The .usect directive reserves space in a specific uninitialized named section.

Each time you invoke the .bss or .usect directive, the assembler reserves additional space in the .bss or the named section. The syntaxes for these directives are:

	<b>.bss</b> <i>symbol</i> , <i>size in bytes</i> [, <i>alignment</i> ], <i>bank offset</i> ]
<i>symbol</i>	<b>.usect</b> "section name", <i>size in bytes</i> [, <i>alignment</i> ], <i>bank offset</i> ]

<i>symbol</i>	points to the first byte reserved by this invocation of the .bss or .usect directive. The <i>symbol</i> corresponds to the name of the variable that you are reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the .global directive).
<i>size in bytes</i>	is an absolute expression. The .bss directive reserves <i>size in bytes</i> bytes in the .bss section. You must specify a size; there is no default value. The .usect directive reserves <i>size in bytes</i> bytes in <i>section name</i> . You must specify a size; there is no default value.
<i>alignment</i>	is an optional parameter. It specifies the minimum alignment in bytes required by the space allocated. The default value is byte aligned. The value must be power of 2.
<i>bank offset</i>	is an optional parameter. It ensures that the space allocated to the symbol occurs on a specific memory bank boundary. The <i>bank offset</i> measures the number of bytes to offset from the alignment specified before assigning the symbol to that location.
<i>section name</i>	tells the assembler which named section to reserve space in. For more information, see <a href="#">Section 2.2.3, Named Sections</a> .

The initialized section directives (`.text`, `.data`, and `.sect`) tell the assembler to stop assembling into the current section and begin assembling into the indicated section. The `.bss` and `.usect` directives, however, *do not* end the current section and begin a new one; they simply escape from the current section temporarily. The `.bss` and `.usect` directives can appear anywhere in an initialized section without affecting its contents. For an example, see [Section 2.2.6, Using Sections Directives](#).

The assembler treats uninitialized subsections (created with the `.usect` directive) in the same manner as uninitialized sections. See [Section 2.2.4](#), for more information on creating subsections.

### 2.2.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in TMS320C6000 memory when the program is loaded. Each initialized section is independently relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section-relative references.

Three directives tell the assembler to place code or data into a section. The syntaxes for these directives are:

```
.text  
.data  
.sect " section name "
```

When the assembler encounters one of these directives, it stops assembling into the current section (acting as an implied end of current section command). It then assembles subsequent code into the designated section until it encounters another `.text`, `.data`, or `.sect` directive.

Sections are built through an iterative process. For example, when the assembler first encounters a `.data` directive, the `.data` section is empty. The statements following this first `.data` directive are assembled into the `.data` section (until the assembler encounters a `.text` or `.sect` directive). If the assembler encounters subsequent `.data` directives, it adds the statements following these `.data` directives to the statements already in the `.data` section. This creates a single `.data` section that can be allocated continuously into memory.

Initialized subsections are created with the `.sect` directive. The assembler treats initialized subsections in the same manner as initialized sections. See [Section 2.2.4](#), for more information on creating subsections.

### 2.2.3 Named Sections

Named sections are sections that *you* create. You can use them like the default `.text`, `.data`, and `.bss` sections, but they are assembled separately.

For example, repeated use of the `.text` directive builds up a single `.text` section in the object file. When linked, this `.text` section is allocated into memory as a single unit. Suppose there is a portion of executable code (perhaps an initialization routine) that you do not want allocated with `.text`. If you assemble this segment of code into a named section, it is assembled separately from `.text`, and you can allocate it into memory separately. You can also assemble initialized data that is separate from the `.data` section, and you can reserve space for uninitialized variables that is separate from the `.bss` section.

Two directives let you create named sections:

- The **`.usect`** directive creates uninitialized sections that are used like the `.bss` section. These sections reserve space in RAM for variables.
- The **`.sect`** directive creates initialized sections, like the default `.text` and `.data` sections, that can contain code or data. The `.sect` directive creates named sections with relocatable addresses.

The syntaxes for these directives are:



```
symbol    .usect "section name", size in bytes [, alignment[, bank offset] ]
          .sect " section name "
```

The *section name* parameter is the name of the section. Section names are significant to 200 characters. You can create up to 32 767 separate named sections. For the .usect and .sect directives, a section name can refer to a subsection; see [Section 2.2.4](#) for details.

Each time you invoke one of these directives with a new name, you create a new named section. Each time you invoke one of these directives with a name that was already used, the assembler assembles code or data (or reserves space) into the section with that name. *You cannot use the same names with different directives.* That is, you cannot create a section with the .usect directive and then try to use the same section with .sect.

## 2.2.4 Subsections

Subsections are smaller sections within larger sections. Like sections, subsections can be manipulated by the linker. Subsections give you tighter control of the memory map. You can create subsections by using the .sect or .usect directive. The syntaxes for a subsection name are:

```
symbol    .usect "section name:subsection name", size in bytes [, alignment[, bank offset] ]
          .sect " section name : subsection name "
```

A subsection is identified by the base section name followed by a colon and the name of the subsection. A subsection can be allocated separately or grouped with other sections using the same base name. For example, you create a subsection called `_func` within the `.text` section:

```
.sect ".text:_func"
```

Using the linker's `SECTIONS` directive, you can allocate `.text:_func` separately, or with all the `.text` sections. See [Section 7.8.1](#), *SECTIONS Directive Syntax*, for an example using subsections.

You can create two types of subsections:

- Initialized subsections are created using the .sect directive. See [Section 2.2.2](#).
- Uninitialized subsections are created using the .usect directive. See [Section 2.2.1](#).

Subsections are allocated in the same manner as sections. See [Section 7.8](#), *The SECTIONS Directive*, for more information.

### 2.2.5 Section Program Counters

The assembler maintains a separate program counter for each section. These program counters are known as *section program counters*, or *SPCs*.

An SPC represents the current address within a section of code or data. Initially, the assembler sets each SPC to 0. As the assembler fills a section with code or data, it increments the appropriate SPC. If you resume assembling into a section, the assembler remembers the appropriate SPC's previous value and continues incrementing the SPC from that value.

The assembler treats each section as if it began at address 0; the linker relocates each section according to its final location in the memory map. For more information, see [Section 2.4, Relocation](#).

### 2.2.6 Using Sections Directives

[Figure 2-2](#) shows how you can build COFF sections incrementally, using the sections directives to swap back and forth between the different sections. You can use sections directives to begin assembling into a section for the first time, or to continue assembling into a section that already contains code. In the latter case, the assembler simply appends the new code to the code that is already in the section.

The format in [Figure 2-2](#) is a listing file. [Figure 2-2](#) shows how the SPCs are modified during assembly. A line in a listing file has four fields:

- |                |   |
|----------------|---|
| <b>Field 1</b> | contains the source code line counter.  |
| <b>Field 2</b> | contains the section program counter.   |
| <b>Field 3</b> | contains the object code.               |
| <b>Field 4</b> | contains the original source statement. |

See [Section 3.10, Source Listings](#) for more information on interpreting the fields in a source listing.

## 2-2 Using Sections Directives Example

```

1          *****
2          ** Assemble an initialized table into .data. **
3          *****
4 00000000          .data
5 00000000 00000011 coeff  .word  011h,022h
6 00000004 00000022
7          *****
8          ** Reserve space in .bss for a variable. **
9          *****
9 00000000          .bss  var1,4
10 00000004          .bss  buffer,40
11          *****
12          ** Still in .data section **
13          *****
14 00000008 00001234 ptr  .word  01234h
15          *****
16          ** Assemble code into .text section **
17          *****
18 00000000          .text
19 00000000 00800528 sum:   MVK      10,A1
20 00000004 021085E0      ZERO    A4
21
22 00000008 01003664 aloop: LDW      *A0++,A2
23 0000000c 00004000      NOP      3
24 00000010 0087E1A0      SUB      A1,1,A1
25 00000014 021041E0      ADD      A2,A4,A4
26 00000018 80000112 [A1]  B        aloop
27 0000001c 00008000      NOP      5
28
29 00000020 0200007C-      STW      A4, *+B14(var1)
30          *****
31          ** Assemble another initialized table in .data **
32          *****
33 0000000c          .data
34 0000000c 000000AA ivals  .word  0aah, 0bbh, 0cch
35 00000010 000000BB
36 00000014 000000CC
37          *****
38          ** Define another section for more variables. **
39          *****
38 00000000          var2  .usect  "newvars",4
39 00000004          inbuf  .usect  "newvars",4
40          *****
41          ** Assemble more code into the .text section. **
42          *****
43 00000024          .text
44 00000024 01003664 xmult: LDW      *A0++,A2
45 00000028 00006000      NOP      4
46 0000002c 020C4480      MPYHL   A2,A3,A4
47 00000030 02800028-      MVKHL   var2,A5
48 00000034 02800068-      MVKH    var2,A5
49 00000038 02140274      STW      A4,*A5
50          *****
51          ** Define a named section for interrupt vectors **
52          *****
53 00000000          .sect  "vectors"
54 00000000 00000012'      B        sum
55 00000004 00008000      NOP      5

```

Field 1    Field 2    Field 3    Field 4

As Figure 2-3 shows, the file in Figure 2-2 creates five sections:

<b>.text</b>	contains 15 32-bit words of object code.
<b>.data</b>	contains six words of initialized data.
<b>vectors</b>	is a named section created with the .sect directive; it contains two words of object code.
<b>.bss</b>	reserves 44 bytes in memory.
<b>newvars</b>	is a named section created with the .usect directive; it contains eight bytes in memory.

The second column shows the object code that is assembled into these sections; the first column shows the source statements that generated the object code.

**Figure 2-3. Object Code Generated by the File in [Figure 2-2](#)**

Line numbers	Object code	Section
19 20 22 23 24 25 26 27 29 44 45 46 47 48 49	00800528 021085E0 01003664 00004000 0087E1A0 021041E0 80000112 00008000 0200007C- 01003664 00006000 020C4480 02800028- 02800068- 02140274	.text
5 5 14 34 34 34	00000011 00000022 00001234 000000AA 000000BB 000000CC	.data
54 54	00000000' 00000024'	vectors
9 10	No data— 44 bytes reserved	.bss
38 39	No data— 8 bytes reserved	newvars

## 2.3 How the Linker Handles Sections

The linker has two main functions related to sections. First, the linker uses the sections in COFF object files as building blocks; it combines input sections (when more than one file is being linked) to create output sections in an executable COFF output module. Second, the linker chooses memory addresses for the output sections.

Two linker directives support these functions:

- The *MEMORY* directive allows you to define the memory map of a target system. You can name portions of memory and specify their starting addresses and their lengths.
- The *SECTIONS* directive tells the linker how to combine input sections into output sections and where to place these output sections in memory.

Subsections allow you to manipulate sections with greater precision. You can specify subsections with the linker's *SECTIONS* directive. If you do not specify a subsection explicitly, then the subsection is combined with the other sections with the same base section name.

It is not always necessary to use linker directives. If you do not use them, the linker uses the target processor's default allocation algorithm described in [Section 7.12, Default Allocation Algorithm](#). When you do use linker directives, you must specify them in a linker command file.

Refer to the following sections for more information about linker command files and linker directives:

[Section 7.5, Linker Command Files](#)

[Section 7.7, The MEMORY Directive](#)

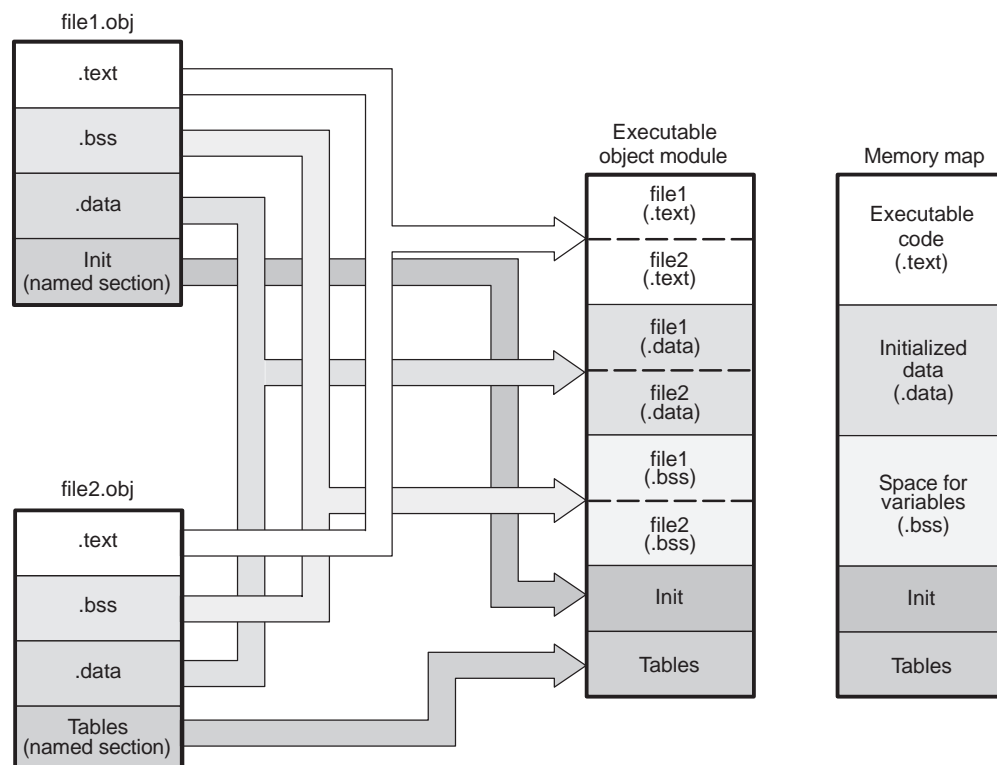
[Section 7.8, The SECTIONS Directive](#)

[Section 7.12, Default Allocation Algorithm](#)

### 2.3.1 Default Memory Allocation

Figure 2-4 illustrates the process of linking two files together.

**Figure 2-4. Combining Input Sections to Form an Executable Object Module**



## Relocation

In [Figure 2-4](#), file1.obj and file2.obj have been assembled to be used as linker input. Each contains the .text, .data, and .bss default sections; in addition, each contains a named section. The executable object module shows the combined sections. The linker combines the .text section from file1.obj and the .text section from file2.obj to form one .text section, then combines the two .data sections and the two .bss sections, and finally places the named sections at the end. The memory map shows how the sections are put into memory; by default, the linker begins at 0h and places the sections one after the other in the following order: .text, .const, .data, .bss, .cinit, and then any named sections in the order they are encountered in the input files.

The C/C++ compiler uses the .const section to store string constants, and variables or arrays that are defined as *far const*. The C/C++ compiler produces tables of data for autoinitializing global variables; these variables are stored in a named section called .cinit (see [Example 7-7](#)). For more information on the .const and .cinit sections, see the *TMS320C6000 Optimizing Compiler User's Guide*.

### 2.3.2 Placing Sections in the Memory Map

[Figure 2-4](#) illustrates the linker's default method for combining sections. Sometimes you may not want to use the default setup. For example, you may not want all of the .text sections to be combined into a single .text section. Or you may want a named section placed where the .data section would normally be allocated. Most memory maps contain various types of memory (RAM, ROM, EPROM, etc.) in varying amounts; you may want to place a section in a specific type of memory.

For further explanation of section placement within the memory map, see the discussions in [Section 7.7](#), *The MEMORY Directive*, and [Section 7.8](#), *The SECTIONS Directive*.

## 2.4 Relocation

The assembler treats each section as if it began at address 0. All relocatable symbols (labels) are relative to address 0 in their sections. Of course, all sections cannot actually begin at address 0 in memory, so the linker *relocates* sections by:

- Allocating them into the memory map so that they begin at the appropriate address as defined with the linker's MEMORY directive
- Adjusting symbol values to correspond to the new section addresses
- Adjusting references to relocated symbols to reflect the adjusted symbol values

The linker uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The linker then uses these entries to patch the references after the symbols are relocated. [Example 2-1](#) contains a code segment for a TMS320C6000 device that generates relocation entries.

#### Example 2-1. Code That Generates Relocation Entries

```

1      .global X
2 00000000 00000012! Z:      B      X      ; Uses an external relocation
3 00000004 0180082A'      MVKL    Y,B3      ; Uses an internal relocation
4 00000008 0180006A'      MVKH    Y,B3      ; Uses an internal relocation
5 0000000C 00004000      NOP      3
6
7 00000010 0001E000 Y:      IDLE
8 00000014 00000212      B      Y
9 00000018 00008000      NOP      5

```

In [Example 2-1](#), both symbols X and Y are relocatable. Y is defined in the .text section of this module; X is defined in another module. When the code is assembled, X has a value of 0 (the assembler assumes all undefined external symbols have values of 0), and Y has a value of 16 (relative to address 0 in the .text section). The assembler generates two relocation entries: one for X and one for Y. The reference to X is an external reference (indicated by the ! character in the listing). The reference to Y is to an internally defined relocatable symbol (indicated by the ' character in the listing).

After the code is linked, suppose that X is relocated to address 0x7100. Suppose also that the .text section is relocated to begin at address 0x7200; Y now has a relocated value of 0x7210. The linker uses the two relocation entries to patch the two references in the object code:

00000012 B X	becomes	0ffffe012
0180082A MVKL Y	becomes	01B9082A
0180006A MVKH Y	becomes	1860006A

Sometimes an expression contains more than one relocatable symbol, or cannot be evaluated at assembly time. In this case, the assembler encodes the entire expression in the object file. After determining the addresses of the symbols, the linker computes the value of the expression as shown in [Example 2-2](#).

### Example 2-2. Simple Assembler Listing

```

1          .global  sym1, sym2
2
3 00000000 00800028%    MVKL    sym2 - sym1, A1

```

The symbols sym1 and sym2 are both externally defined. Therefore, the assembler cannot evaluate the expression sym2 - sym1, so it encodes the expression in the object file. The '%' listing character indicates a relocation expression. Suppose the linker relocates sym2 to 300h and sym1 to 200h. Then the linker computes the value of the expression to be 300h - 200h = 100h. Thus the MVKL instruction is patched to:

```
00808028          MVKL    100h, A1
```

#### Expression Cannot Be Larger Than Space Reserved

**Note:** If the value of an expression is larger, in bits, than the space reserved for it, you will receive an error message from the linker.

Each section in a COFF object file has a table of relocation entries. The table contains one relocation entry for each relocatable reference in the section. The linker usually removes relocation entries after it uses them. This prevents the output file from being relocated again (if it is relinked or when it is loaded). A file that contains no relocation entries is an *absolute* file (all its addresses are absolute addresses). If you want the linker to retain relocation entries, invoke the linker with the -r option (see [Section 3.3](#)).

## 2.5 Run-Time Relocation

At times you may want to load code into one area of memory and run it in another. For example, you may have performance-critical code in an external-memory-based system. The code must be loaded into external memory, but it would run faster in internal memory.

The linker provides a simple way to handle this. Using the `SECTIONS` directive, you can optionally direct the linker to allocate a section twice: first to set its load address and again to set its run address. Use the *load* keyword for the load address and the *run* keyword for the run address.

The load address determines where a loader places the raw data for the section. Any references to the section (such as references to labels in it) refer to its run address. The application must copy the section from its load address to its run address before the first reference of the symbol is encountered at run time; this does *not* happen automatically simply because you specify a separate run address. For an example that illustrates how to move a block of code at run time, see [Example 7-9](#).

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is actually allocated as if it were two separate sections of the same size.

Uninitialized sections (such as `.bss`) are not loaded, so the only significant address is the run address. The linker allocates uninitialized sections only once; if you specify both run and load addresses, the linker warns you and ignores the load address.

For a complete description of run-time relocation, see [Section 7.9](#), *Specifying a Section's Run-Time Address*.

## 2.6 Loading a Program

The linker produces executable COFF object modules. An executable object file has the same COFF format as object files that are used as linker input; the sections in an executable object file, however, are combined and relocated into target memory.

To run a program, the data in the executable object module must be transferred, or loaded, into target system memory. Several methods can be used for loading a program, depending on the execution environment. Three common situations are described below:

- Code Composer Studio can load an executable COFF file into a simulator or onto hardware. The Code Composer Studio loader reads the executable file and copies the program into target memory.
- You can use the hex conversion utility (hex6x, which is shipped as part of the assembly language package) to convert the executable COFF object module into one of several object file formats. You can then use the converted file with an EPROM programmer to burn the program into an EPROM.
- A standalone simulator can be invoked by the `load6x` command and the name of the executable object file. The standalone simulator reads the executable file, copies the program into the simulator and executes it, displaying any C I/O.



## 2.7 Symbols in a COFF File

A COFF file contains a symbol table that stores information about symbols in the program. The linker uses this table when it performs relocation. Debugging tools can also use the symbol table to provide symbolic debugging.

### 2.7.1 External Symbols

External symbols are symbols that are defined in one module and referenced in another module. You can use the `.def`, `.ref`, or `.global` directive to identify symbols as external:

<b><code>.def</code></b>	The symbol is defined in the current module and used in another module.
<b><code>.ref</code></b>	The symbol is referenced in the current module, but defined in another module.
<b><code>.global</code></b>	The symbol may be either of the above.

The following code segment illustrates these definitions.

```
.def      x
.ref      y
.global   z
.global   q

q:  B      B3
    NOP    4
    MVK    1, B1
x:  MV      A0, A1
    MVK    y, B3
    MVKH   y, B3
    B      z
    NOP    5
```

In this example, the `.def` definition of `x` says that it is an external symbol defined in this module and that other modules can reference `x`. The `.ref` definition of `y` says that it is an undefined symbol that is defined in another module. The `.global` definition of `z` says that it is defined in some module and available in this file. The `.global` definition of `q` says that it is defined in this module and that other modules can reference `q`.

The assembler places `x`, `y`, `z`, and `q` in the object file's symbol table. When the file is linked with other object files, the entries for `x` and `q` resolve references to `x` and `q` in other files. The entries for `y` and `z` cause the linker to look through the symbol tables of other files for `y`'s and `z`'s definitions.

The linker must match all references with corresponding definitions. If the linker cannot find a symbol's definition, it prints an error message about the unresolved reference. This type of error prevents the linker from creating an executable object module.

### 2.7.2 The Symbol Table

The assembler always generates an entry in the symbol table when it encounters an external symbol (both definitions and references defined by one of the directives in [Section 2.7.1](#)). The assembler also creates special symbols that point to the beginning of each section; the linker uses these symbols to relocate references to other symbols.

The assembler does not usually create symbol table entries for any symbols other than those described above, because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with the `.global` directive. For informational purposes, it is sometimes useful to have entries in the symbol table for each symbol in a program. To accomplish this, invoke the assembler with the `-as` option (see [Section 3.3](#)).

*Symbols in a COFF File*

---

## Assembler Description

The TMS320C6000™ assembler translates assembly language source files into machine language object files. These files are in common object file format (COFF), which is discussed in [Chapter 2, Introduction to Common Object File Format](#), and [Appendix A, Common Object File Format](#). Source files can contain the following assembly language elements:

Assembler directives	described in <a href="#">Chapter 4</a>
Macro directives	described in <a href="#">Chapter 5</a>
Assembly language instructions	described in the <i>TMS320C6000 CPU and Instruction Set Reference Guide</i>

Topic	Page
<b>3.1 Assembler Overview.....</b>	<b><a href="#">36</a></b>
<b>3.2 The Assembler's Role in the Software Development Flow .....</b>	<b><a href="#">36</a></b>
<b>3.3 Invoking the Assembler .....</b>	<b><a href="#">38</a></b>
<b>3.4 Naming Alternate Directories for Assembler Input .....</b>	<b><a href="#">40</a></b>
<b>3.5 Source Statement Format .....</b>	<b><a href="#">42</a></b>
<b>3.6 Constants .....</b>	<b><a href="#">45</a></b>
<b>3.7 Character Strings .....</b>	<b><a href="#">47</a></b>
<b>3.8 Symbols.....</b>	<b><a href="#">47</a></b>
<b>3.9 Expressions .....</b>	<b><a href="#">54</a></b>
<b>3.10 Source Listings.....</b>	<b><a href="#">57</a></b>
<b>3.11 Debugging Assembly Source .....</b>	<b><a href="#">58</a></b>
<b>3.12 Cross-Reference Listings.....</b>	<b><a href="#">60</a></b>

### 3.1 Assembler Overview

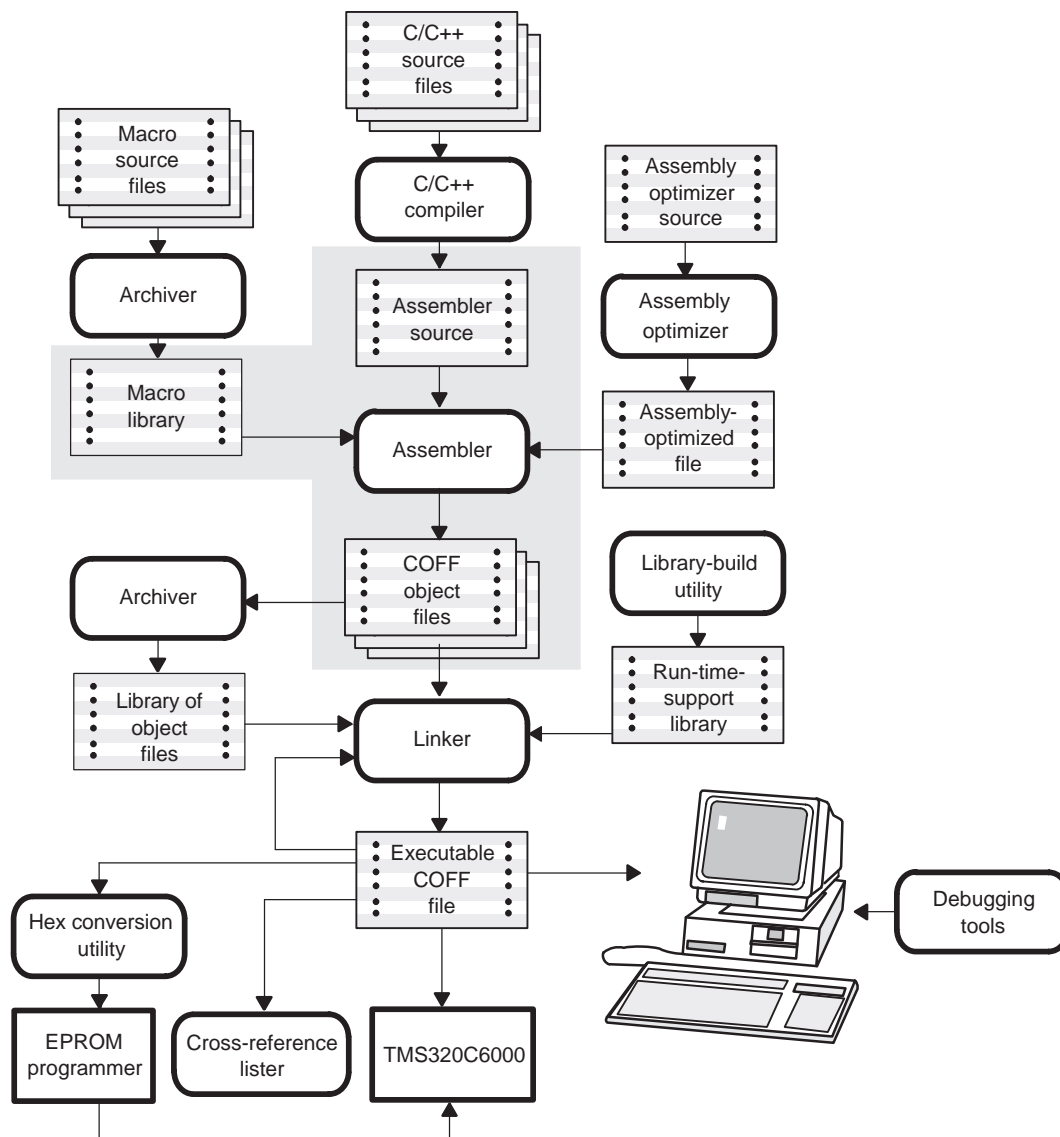
The 2-pass assembler does the following:

- Processes the source statements in a text file to produce a relocatable object file
- Produces a source listing (if requested) and provides you with control over this listing
- Allows you to segment your code into sections and maintain a section program counter (SPC) for each section of object code
- Defines and references global symbols and appends a cross-reference listing to the source listing (if requested)
- Allows conditional assembly
- Supports macros, allowing you to define macros inline or in a library

### 3.2 The Assembler's Role in the Software Development Flow

[Figure 3-1](#) illustrates the assembler's role in the software development flow. The shaded portion highlights the most common assembler development path. The assembler accepts assembly language source files as input, both those you create and those created by the TMS320C6000 C/C++ compiler.

**Figure 3-1. The Assembler in the TMS320C6000 Software Development Flow**



### 3.3 Invoking the Assembler

To invoke the assembler, enter the following:

**cl6x** *input file* [*options*]

**cl6x** is the command that invokes the assembler through the compiler. The compiler considers any file with an .asm extension to be an assembly file and calls the assembler.

*input file* names the assembly language source file.

*options* identify the assembler options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen.

The valid assembler options are as follows:

**-@** **-@=*filename*** appends the contents of a file to the command line. You can use this option to avoid limitations on command line length imposed by the host operating system. Use an asterisk or a semicolon (\* or ;) at the beginning of a line in the command file to include comments. Comments that begin in any other column must begin with a semicolon. Within the command file, filenames or option parameters containing embedded spaces or hyphens must be surrounded with quotation marks. For example: "this-file.asm"

**-aa** creates an absolute listing. When you use -aa, the assembler does not produce an object file. The -aa option is used in conjunction with the absolute lister.

**-ac** makes case insignificant in the assembly language files. For example, -ac makes the symbols ABC and abc equivalent. *If you do not use this option, case is significant* (default). Case significance is enforced primarily with symbol names, not with mnemonics and register names.

**-ad** **-ad=*name* [=*value*]** sets the *name* symbol. This is equivalent to inserting *name* .set [*value*] at the beginning of the assembly file. If *value* is omitted, the symbol is set to 1. For more information, see [Section 3.8.4, Defining Symbolic Constants \(-ad Option\)](#).

**-ahc** **-ahc=*filename*** tells the assembler to copy the specified file for the assembly module. The file is inserted before source file statements. The copied file appears in the assembly listing files.

**-ahi** **-ahi=*filename*** tells the assembler to include the specified file for the assembly module. The file is included before source file statements. The included file does not appear in the assembly listing files.

**-al** (lowercase L) produces a listing file with the same name as the input file with a .lst extension.

**-apd** performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a .ppa extension.

**-api** performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of files included with the .include directive. The list is written to a file with the same name as the source file but with a .ppa extension.

---

<b>-as</b>	puts all defined symbols in the object file's symbol table. The assembler usually puts only global symbols into the symbol table. When you use -as, symbols defined as labels or as assembly-time constants are also placed in the table.
<b>-au</b>	<b>-au=name</b> undefines the predefined constant <i>name</i> , which overrides any -ad options for the specified constant.
<b>-ax</b>	produces a cross-reference table and appends it to the end of the listing file; it also adds cross-reference information to the object file for use by the cross-reference utility. If you do not request a listing file but use the -ax option, the assembler creates a listing file automatically, naming it with the same name as the input file with a .lst extension.
<b>-g</b>	(--symdebug:dwarf) enables assembler source debugging in the C source debugger. Line information is output to the COFF file for every line of source in the assembly language source file. You cannot use the -g option on assembly code that contains .line directives. See <a href="#">Section 3.11</a> , <i>Debugging Assembly Source</i> , for more information.
<b>-I</b>	specifies a directory where the assembler can find files named by the .copy, .include, or .mlib directives. The format of the -I option is -I=pathname. There is no limit to the number of directories you can specify in this manner; each pathname must be preceded by the -I option. For more information, see <a href="#">Section 3.4.1</a> , <i>Using the -I Assembler Option</i> .
<b>-me</b>	produces object code in big-endian format.
<b>-mv</b>	<b>-mv=num</b> selects the target CPU version using the last four digits of the TMS320C6000 part number.
<b>--no_compress</b>	prevents compression (C6400+). Fetch packets that utilize 16-bit instructions are not created.
<b>--no_reload_errors</b>	turns off all reload-related loop buffer error messages (C6400+)
<b>-q</b>	suppresses the banner and progress information (assembler runs in quiet mode).

For more information about the -me and -mv options, see the *TMS320C6000 Optimizing C Compiler User's Guide*.

### 3.4 Naming Alternate Directories for Assembler Input

The `.copy`, `.include`, and `.mlib` directives tell the assembler to use code from external files. The `.copy` and `.include` directives tell the assembler to read source statements from another file, and the `.mlib` directive names a library that contains macro functions. [Chapter 4, Assembler Directives](#), contains examples of the `.copy`, `.include`, and `.mlib` directives. The syntax for these directives is:

```
.copy ["filename"]
.include ["filename"]
.mlib ["filename"]
```

The *filename* names a copy/include file that the assembler reads statements from or a macro library that contains macro definitions. If *filename* begins with a number the double quotes are required. The filename may be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in the following locations in the order given:

1. The directory that contains the current source file. The current source file is the file being assembled when the `.copy`, `.include`, or `.mlib` directive is encountered.
2. Any directories named with the `-I` assembler option
3. Any directories named with the `C6X_A_DIR` or `A_DIR` environment variable

Because of this search hierarchy, you can augment the assembler's directory search algorithm by using the `-I` assembler option (described in [Section 3.4.1](#)) or the `C6X_A_DIR` or `A_DIR` environment variable (described in [Section 3.4.2](#)).

#### 3.4.1 Using the `-I` Assembler Option

The `-I` assembler option names an alternate directory that contains copy/ include files or macro libraries. The format of the `-I` option is as follows:

```
cl6x -I=pathname source filename [other options]
```

There is no limit to the number of `-I` options per invocation; each `-I` option names one pathname. In assembly source, you can use the `.copy`, `.include`, or `.mlib` directive without specifying path information. If the assembler does not find the file in the directory that contains the current source file, it searches the paths designated by the `-I` options.

For example, assume that a file called `source.asm` is in the current directory; `source.asm` contains the following directive statement:

```
.copy "copy.asm"
```

Assume the following paths for the `copy.asm` file:

```
UNIX™:                /tools/files/copy.asm
Windows™:             c:\tools\files\copy.asm
```

You could set up the search path with the commands shown below:

Operating System	Enter
UNIX (Bourne shell)	<code>cl6x -I/tools/files source.asm</code>
Windows	<code>cl6x -Ic:\tools\files source.asm</code>

The assembler first searches for `copy.asm` in the current directory because `source.asm` is in the current directory. Then the assembler searches in the directory named with the `-I` option.



### 3.4.2 Using the C6X\_A\_DIR or A\_DIR Environment Variable

An environment variable is a system symbol that you define and assign a string to. The assembler uses the C6X\_A\_DIR and A\_DIR environment variables to name alternate directories that contain copy/include files or macro libraries.

The assembler looks for the C6X\_A\_DIR environment variable first and then reads and processes it. If it does not find this variable, it reads the A\_DIR environment variable and processes it. If both variables are set, the settings of the processor-specific variable are used. The processor-specific variable is useful when you are using Texas Instruments tools for different processors at the same time.

If the assembler does not find C6X\_A\_DIR and A\_DIR, it then searches for C6X\_C\_DIR and C\_DIR. See the *TMS320C6000 Optimizing Compiler User's Guide* for details on C6X\_C\_DIR and C\_DIR.

The command syntax for assigning the environment variable is as follows:

Operating System	Enter
UNIX (Bourne Shell)	<b>A_DIR="pathname<sub>1</sub>;pathname<sub>2</sub>; . . . "; export A_DIR</b>
Windows	<b>set A_DIR=pathname<sub>1</sub>;pathname<sub>2</sub>; . . .</b>

The *pathnames* are directories that contain copy/include files or macro libraries. The pathnames must follow these constraints:

- Pathnames must be separated with a semicolon.
- Spaces or tabs at the beginning or end of a path are ignored. For example the space before and after the semicolon in the following is ignored:

```
set A_DIR= c:\path\one\to\tools ; c:\path\two\to\tools
```

- Spaces and tabs are allowed within paths to accommodate Windows directories that contain spaces. For example, the pathnames in the following are valid:

```
set a_DIR=c:\first path\to\tools;d:\second path\to\tools
```

In assembly source, you can use the .copy, .include, or .mlib directive without specifying path information. If the assembler does not find the file in the directory that contains the current source file or in directories named by the -I option, it searches the paths named by the environment variable.

For example, assume that a file called source.asm contains these statements:

```
.copy "copy1.asm"
.copy "copy2.asm"
```

Assume the following paths for the files:

UNIX: /tools/files/copy1.asm and /dsys/copy2.asm

Windows: c:\tools\files\copy1.asm and c:\dsys\copy2.asm

You could set up the search path with the commands shown below:

Operating System	Enter
UNIX (Bourne shell)	<b>A_DIR="/dsys"; export A_DIR cl6x -I=/tools/files source.asm</b>
Windows	<b>set A_DIR=c:\dsys cl6x -Ic:\tools\files source.asm</b>

The assembler first searches for copy1.asm and copy2.asm in the current directory because source.asm is in the current directory. Then the assembler searches in the directory named with the -I option and finds copy1.asm. Finally, the assembler searches the directory named with A\_DIR and finds copy2.asm.

The environment variable remains set until you reboot the system or reset the variable by entering one of these commands:

## Source Statement Format

Operating System	Enter
UNIX (Bourne shell)	set A_DIR=
Windows	unset A_DIR

### 3.5 Source Statement Format

TMS320C6000 assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. A source statement can contain five ordered fields (label, mnemonic, unit specifier, operand list, and comment). The general syntax for source statements is as follows:

`[label[:]] [||] [[register]] mnemonic [unit specifier] [operand list][;comment]`

Following are examples of source statements:

```
two      .set  2      ; Symbol Two = 2
Label:   MVK     two,A2 ; Move 2 into register A2
         .word 016h   ; Initialize a word with 016h
```

The C6000 assembler reads up to 200 characters per line. Any characters beyond 200 are truncated. Keep the operational part of your source statements (that is, everything other than comments) less than 200 characters in length for correct assembly. Your comments can extend beyond the 200-character limit, but the truncated portion is not included in the listing file.

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. Tab and space characters are blanks. You must separate the operand list from the preceding field with a blank.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (\* or ;), but comments that begin in any other column *must* begin with a semicolon.
- In a conditional instruction, the condition register must be surrounded by square brackets.
- The functional unit specifier is optional. If you do not specify the functional unit, the assembler assigns a legal functional unit based on the mnemonic field.
- A mnemonic cannot begin in column 1 or it will be interpreted as a label.

The following sections describe each of the fields.

### 3.5.1 Label Field

Labels are optional for all assembly language instructions and for most (but not all) assembler directives. When used, a label must begin in column 1 of a source statement. A label can contain up to 128 alphanumeric characters (A-Z, a-z, 0-9, \_, and \$). Labels are case sensitive (except when the -ac option is used), and the first character cannot be a number. A label can be followed by a colon (:). The colon is not treated as part of the label name. If you do not use a label, the first character position must contain a blank, a semicolon, or an asterisk. You cannot use a label with an instruction that is in parallel with a previous instruction.

When you use a label, its value is the current value of the SPC. The label points to the statement it is associated with. For example, if you use the .word directive to initialize several words, a label points to the first word. In the following example, the label Start has the value 40h.

```
.      .      .      .
.      .      .      .
.      .      .      .

      9                      * Assume some code was assembled
10 00000040 0000000A Start: .word 0Ah,3,7
      00000044 00000003
      00000048 00000007
```

A label on a line by itself is a valid statement. The label assigns the current value of the section program counter to the label; this is equivalent to the following directive statement:

```
label .equ $ ; $ provides the current value of the SPC
```

When a label appears on a line by itself, it points to the instruction on the next line (the SPC is not incremented):

```
1 00000000          Here:
2 00000000 00000003          .word 3
```

If you do not use a label, the character in column 1 must be a blank, an asterisk, or a semicolon.

### 3.5.2 Mnemonic Field

The mnemonic field follows the label field. The mnemonic field cannot start in column 1; if it does, it is interpreted as a label. There is one exception: the parallel bars (||) of the mnemonic field can start in column 1. The mnemonic field can begin with one of the following items:

- Pipe symbols (||) indicate instructions that are in parallel with a previous instruction. You can have up to eight instructions running in parallel. The following example demonstrates six instructions running in parallel:

```

          Inst1
||      Inst2
||      Inst3
||      Inst4
||      Inst5
||      Inst6
||      Inst7
          } These five instructions run in
           parallel with the first instruc-
           tion.
```

- Square brackets ( [ ] ) indicate conditional instructions. The machine-instruction mnemonic is executed based on the value of the register within the brackets; valid register names are A0 for C64xx only, A1, A2, B0, B1, and B2. The instruction is executed if the value of the register is nonzero. If the register name is preceded by an exclamation point (!), then the instruction is executed if the value of the register is 0. For example:

```
[A1] ZERO A2 ; If A1 is not equal to zero, A2 = 0
```

Next, the mnemonic field contains one of the following items:

- Machine-instruction mnemonic (such as ADDK, MVKH, B)
- Assembler directive (such as .data, .list, .equ)
- Macro directive (such as .macro, .var, .mexit)
- Macro call

### 3.5.3 Unit Specifier Field

The unit specifier field is an optional field that follows the mnemonic field for machine-instruction mnemonics. The unit specifier field begins with a period (.) followed by a functional unit specifier. In general, one instruction can be assigned to each functional unit in a single instruction cycle. There are eight functional units, two of each functional type:

<b>.D1</b> and <b>.D2</b>	Data/addition/subtraction
<b>.L1</b> and <b>.L2</b>	ALU/compares/long data arithmetic
<b>.M1</b> and <b>.M2</b>	Multiply
<b>.S1</b> and <b>.S2</b>	Shift/ALU/branch/bit field

ALU refers to an arithmetic logic unit.

There are several ways to use the unit specifier field:

- You can specify the particular functional unit (for example, .D1).
- You can specify only the functional type (for example, .M), and the assembler assigns the specific unit (for example, .M2).
- If you do not specify the functional unit, the assembler assigns the functional unit based on the mnemonic field and operand field.

For more information on functional units, including which assembly instructions require which functional type, see the *TMS320C6000 CPU and Instruction Set Reference Guide*.

### 3.5.4 Operand Field

The operand field follows the mnemonic field and contains one or more operands. The operand field is not required for all instructions or directives. An operand consists of the following items:

- Symbols (see [Section 3.8](#))
- Constants (see [Section 3.6](#))
- Expressions (combination of constants and symbols; see [Section 3.9](#))

You must separate operands with commas.

### 3.5.5 Comment Field

A comment can begin in any column and extends to the end of the source line. A comment can contain any ASCII character, including blanks. Comments are printed in the assembly source listing, but they do not affect the assembly.

A source statement that contains only a comment is valid. If it begins in column 1, it can start with a semicolon ( ; ) or an asterisk ( \* ). Comments that begin anywhere else on the line must begin with a semicolon. The asterisk identifies a comment only if it appears in column 1.

## 3.6 Constants

The assembler supports six types of constants:

- Binary integer
- Octal integer
- Decimal integer
- Hexadecimal integer
- Character
- Assembly-time

The assembler maintains each constant internally as a 32-bit quantity. Constants are not sign extended. For example, the constant 00FFh is equal to 00FF (base 16) or 255 (base 10); it *does not* equal -1. However, when used with the .byte directive, -1 is equivalent to 00FFh.

### 3.6.1 Binary Integers

A binary integer constant is a string of up to 32 binary digits (0s and 1s) followed by the suffix B (or b). If fewer than 32 digits are specified, the assembler right justifies the value and fills the unspecified bits with zeros. These are examples of valid binary constants:

00000000B	Constant equal to 0 <sub>10</sub> or 0 <sub>16</sub>
0100000b	Constant equal to 32 <sub>10</sub> or 20 <sub>16</sub>
01b	Constant equal to 1 <sub>10</sub> or 1 <sub>16</sub>
11111000B	Constant equal to 248 <sub>10</sub> or 0F8 <sub>16</sub>

### 3.6.2 Octal Integers

An octal integer constant is a string of up to 11 octal digits (0 through 7) followed by the suffix Q (or q). These are examples of valid octal constants:

10Q	Constant equal to 8 <sub>10</sub> or 8 <sub>16</sub>
010	Constant equal to 8 <sub>10</sub> or 8 <sub>16</sub> © format)
100000Q	Constant equal to 32 768 <sub>10</sub> or 8000 <sub>16</sub>
226q	Constant equal to 150 <sub>10</sub> or 96 <sub>16</sub>

### 3.6.3 Decimal Integers

A decimal integer constant is a string of decimal digits ranging from -2147 483 648 to 4 294 967 295. These are examples of valid decimal constants:

1000	Constant equal to 1000 <sub>10</sub> or 3E8 <sub>16</sub>
-32768	Constant equal to -32 768 <sub>10</sub> or 8000 <sub>16</sub>
25	Constant equal to 25 <sub>10</sub> or 19 <sub>16</sub>

## Constants

---

### 3.6.4 Hexadecimal Integers

A hexadecimal integer constant is a string of up to eight hexadecimal digits followed by the suffix H (or h). Hexadecimal digits include the decimal values 0-9 and the letters A-F or a-f. *A hexadecimal constant must begin with a decimal value (0-9).* If fewer than eight hexadecimal digits are specified, the assembler right justifies the bits. These are examples of valid hexadecimal constants:

78h	Constant equal to $120_{10}$ or $0078_{16}$
0x78	Constant equal to $120_{10}$ or $0078_{16}$ © format)
0Fh	Constant equal to $15_{10}$ or $000F_{16}$
37ACh	Constant equal to $14\,252_{10}$ or $37AC_{16}$

### 3.6.5 Character Constants

A character constant is a single character enclosed in *single* quotes. The characters are represented internally as 8-bit ASCII characters. Two consecutive single quotes are required to represent each single quote that is part of a character constant. A character constant consisting only of two single quotes is valid and is assigned the value 0. These are examples of valid character constants:

'a'	Defines the character constant <i>a</i> and is represented internally as $61_{16}$
'C'	Defines the character constant <i>C</i> and is represented internally as $43_{16}$
''	Defines the character constant <i>'</i> and is represented internally as $27_{16}$
"	Defines a null character and is represented internally as $00_{16}$

Notice the difference between character *constants* and character *strings* (Section 3.7 discusses character strings). A character constant represents a single integer value; a string is a sequence of characters.

### 3.6.6 Assembly-Time Constants

If you use the .set directive to assign a value to a symbol (see [Define Assembly-Time Constant](#)), the symbol becomes a constant. To use this constant in expressions, the value that is assigned to it must be absolute. For example:

```
sym    .set 3
      MVK    sym,B1
```

You can also use the .set directive to assign symbolic constants for register names. In this case, the symbol becomes a synonym for the register:

```
sym    .set B1
      MVK    10,sym
```

## 3.7 Character Strings

A character string is a string of characters enclosed in *double* quotes. Double quotes that are part of character strings are represented by two consecutive double quotes. The maximum length of a string varies and is defined for each directive that requires a character string. Characters are represented internally as 8-bit ASCII characters.

These are examples of valid character strings:

**"sample program"** defines the 14-character string *sample program*.

**"PLAN "C"'"** defines the 8-character string *PLAN "C"*.

Character strings are used for the following:

- Filenames, as in .copy "filename"
- Section names, as in .sect "section name"
- Data initialization directives, as in .byte "charstring"
- Operands of .string directives

## 3.8 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 200 alphanumeric characters (A-Z, a-z, 0-9, \$, and \_). The first character in a symbol cannot be a number, and symbols cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler recognizes ABC, Abc, and abc as three unique symbols. You can override case sensitivity with the -ac assembler option (see [Section 3.3](#)). A symbol is valid only during the assembly in which it is defined, unless you use the .global directive or the .def directive to declare it as an external symbol (see [Identify Global Symbols](#) ).

### 3.8.1 Labels

Symbols used as labels become symbolic addresses that are associated with locations in the program. Labels used locally within a file must be unique. Mnemonic opcodes and assembler directive names without the . prefix are valid label names.

Labels can also be used as the operands of .global, .ref, .def, or .bss directives; for example:

```
.global label1

label2: MVKL    label2, B3
        MVKH    label2, B3
        B       label1
        NOP     5
```

### 3.8.2 Local Labels

Local labels are special labels whose scope and effect are temporary. A local label can be defined in two ways:

- \$n, where n is a decimal digit in the range 0-9. For example, \$4 and \$1 are valid local labels. See [Example 3-1](#).
- name?, where name is any legal symbol name as described above. The assembler replaces the question mark with a period followed by a unique number. When the source code is expanded, *you will not see the unique number in the listing file*. Your label appears with the question mark as it did in the source definition. You cannot declare this label as global. See [Example 3-2](#).

Normal labels must be unique (they can be declared only once), and they can be used as constants in the operand field. Local labels, however, can be undefined and defined again. Local labels cannot be defined by directives.

## Symbols

---

A local label can be undefined or reset in one of these ways:

- By using the `.newblock` directive
- By changing sections (using a `.sect`, `.text`, or `.data` directive)
- By entering an include file (specified by the `.include` or `.copy` directive)
- By leaving an include file (specified by the `.include` or `.copy` directive)

### **Example 3-1. Local Labels of the Form \$n**

This is an example of code that declares and uses a local label legally:

```
$1:
    SUB    A1,1,A1
[A1] B    $1
    SUBC   A3,A0,A3
    NOP    4

    .newblock      ; undefine $1 to use it again

$1 SUB    A2,1,A2
[A2] B    $1
    MPY    A3,A3,A3
    NOP    4
```

The following code uses a local label illegally:

```
$1:
    SUB    A1,1,A1
[A1] B    $1
    SUBC   A3,A0,A3
    NOP    4
$1 SUB    A2,1,A2    ; WRONG - $1 is multiply defined
[A2] B    $1
    MPY    A3,A3,A3
    NOP    4
```

The \$1 label is not undefined before being reused by the second branch instruction. Therefore, \$1 is redefined, which is illegal.

Local labels are especially useful in macros. If a macro contains a normal label and is called more than once, the assembler issues a multiple-definition error. If you use a local label and `.newblock` within a macro, however, the local label is used and reset each time the macro is expanded.

Up to ten local labels can be in effect at one time. After you undefine a local label, you can define it and use it again. Local labels do not appear in the object code symbol table.

Because local labels are intended to be used only locally, branches to local labels are not expanded in case the branch's offset is out of range.



### Example 3-2. Local Labels of the Form name?

```

*****
** First definition of local label mylab                **
*****
        nop
mylab?  nop
        B mylab?
        nop 5
*****
** Include file has second definition of mylab          **
*****
        .copy  "a.inc"
*****
** Third definition of mylab, reset upon exit from .include **
*****
mylab?  nop
        B mylab?
        nop 5
*****
** Fourth definition of mylab in macro, macros use different **
** namespace to avoid conflicts                             **
*****
mymac   .macro
mylab?  nop
        B mylab?
        nop 5
        .endm
*****
** Macro invocation                                         **
*****
        mymac
*****
** Reference to third definition of mylab. Definition is not **
** reset by macro invocation.                               **
*****
        B mylab?
        nop 5
*****
** Changing section, allowing fifth definition of mylab     **
*****
        .sect  "Sect_One"
        nop
mylab?  .word 0
        nop
        B mylab?
        nop 5
*****
** The .newblock directive allows sixth definition of mylab **
*****
        .newblock
mylab?  .word 0
        nop
        nop
        B mylab?
        nop 5

```

## Symbols

### 3.8.3 Symbolic Constants

Symbols can be set to constant values. By using constants, you can equate meaningful names with constant values. The `.set` and `.struct/.tag/.endstruct` directives enable you to set constants to symbolic names. Symbolic constants *cannot* be redefined. The following example shows how these directives can be used:

```
K      .set  1024          ; constant definitions
maxbuf .set  2*K

item   .struct             ; item structure definition
value  .int                ; value offset = 0
delta  .int                ; delta offset = 4
i_len  .endstruct          ; item size    = 8

array  .tag  item
       .bss  array, i_len*K ; declare an array of K "items"
       .text
       LDW   *+B14(array.delta + 2*i_len),A1
                               ; access array [2].delta
```

The assembler also has several predefined symbolic constants; these are discussed in [Section 3.8.5](#).

### 3.8.4 Defining Symbolic Constants (-ad Option)

The `-ad` option equates a constant value or a string with a symbol. The symbol can then be used in place of a value in assembly source. The format of the `-ad` option is as follows:

**cl6x -ad=name[=value]**

The *name* is the name of the symbol you want to define. The *value* is the constant or string value you want to assign to the symbol. If the *value* is omitted, the symbol is set to 1. If you want to define a quoted string and keep the quotation marks, do one of the following:

- For Windows®, use **-ad=name="\value"**. For example, `-ad=car="\sedan\"`
- For UNIX®, use **-ad=name='"value'"**. For example, `-ad=car="sedan"`
- For Code Composer Studio, enter the definition in a file and include that file with the `-@` option.

Once you have defined the name with the `-ad` option, the symbol can be used in place of a constant value, a well-defined expression, or an otherwise undefined symbol used with assembly directives and instructions. For example, on the command line you enter:

```
cl6x -adSYM1=1 -adSYM2=2 -adSYM3=3 -adSYM4=4 value.asm
```

Since you have assigned values to SYM1, SYM2, SYM3, and SYM4, you can use them in source code. [Example 3-3](#) shows how the `value.asm` file uses these symbols without defining them explicitly.

### Example 3-3. Using Symbolic Constants Defined on Command Line

```

If_4:  .if      SYM4 = SYM2 * SYM2
        .byte   SYM4           ; Equal values
        .else
        .byte   SYM2 * SYM2    ; Unequal values
        .endif

IF_5:  .if      SYM1 <= 10
        .byte   10            ; Less than / equal
        .else
        .byte   SYM1          ; Greater than
        .endif

IF_6:  .if      SYM3 * SYM2 != SYM4 + SYM2
        .byte   SYM3 * SYM2    ; Unequal value
        .else
        .byte   SYM4 + SYM4    ; Equal values
        .endif

IF_7:  .if      SYM1 = SYM2
        .byte   SYM1
        .elseif SYM2 + SYM3 = 5
        .byte   SYM2 + SYM3
        .endif

```

Within assembler source, you can test the symbol defined with the -ad option with the following directives:

Type of Test	Directive Usage
Existence	<code>.if \$isdefed(" name ")</code>
Nonexistence	<code>.if \$isdefed(" name ") = 0</code>
Equal to value	<code>.if name = value</code>
Not equal to value	<code>.if name != value</code>

The argument to the \$isdefed built-in function must be enclosed in quotes. The quotes cause the argument to be interpreted literally rather than as a substitution symbol.

### 3.8.5 Predefined Symbolic Constants

The assembler has several predefined symbols, including the following types:

- **\$**, the dollar-sign character, represents the current value of the section program counter (SPC). \$ is a relocatable symbol.
- **Register symbols**, including A0-A15 and B0-B15 for C6200 and C6700; and A16-31 and B16-31 for C6400 and C6400+.
- **CPU control registers**, including those listed in [Table 3-1](#). Control registers can be entered as all upper-case or all lower-case characters; for example, CSR can also be entered as csr.
- **Processor symbols**, including those listed in [Table 3-2](#).
- **Assembler Version Symbols**. See [Table 3-3](#).

## Symbols

**Table 3-1. CPU Control Registers**

Register	Description
AMR	Addressing mode register
CSR	Control status register
DESR	(C6700+ only) dMAX event status register
DETR	(C6700+ only) dMAX event trigger register
DNUM	(C6400+ only) DSP core number register
ECR	(C6400+ only) Exception clear register
EFR	(C6400+ only) Exception flag register
FADCR	(C6700 only) Floating-point adder configuration register
FAUCR	(C6700 only) Floating-point auxiliary configuration register
FMCR	(C6700 only) Floating-point multiplier configuration register
GFPGFR	(C6400 only) Galois field polynomial generator function register
GPLYA	(C6400+ only) GMPY A-side polynomial register
GPLYB	(C6400+ only) GMPY B-side polynomial register
ICR	Interrupt clear register
IER	Interrupt enable register
IERR	(C6400+ only) Interrupt exception report register
IFR	Interrupt flag register
ILC	(C6400+ only) Inner loop count register
NRP	Nonmaskable interrupt return pointer
IRP	Interrupt return pointer
ISR	Interrupt set register
ITSR	(C6400+ only) Interrupt task state register
ISTP	Interrupt service table pointer
NTSR	(C6400+ only) NMI/Exception task state register
PCE1	Program counter
REP	(C6400+ only) Restricted entry point address register
RILC	(C6400+ only) Reload inner loop count register
SSR	(C6400+ only) Saturation status register
TSCH	(C6400+ only) Time-stamp counter (high 32) register
TSCL	(C6400+ only) Time-stamp counter (low 32) register
TSR	(C6400+ only) Task status register

**Table 3-2. Processor Symbols**

Symbol name	Description
.TMS320C6000	Always set to 1
.TMS320C6200	Set to 1 for C6200, otherwise 0
.TMS320C6400	Set to 1 for C6400 and C6400+, otherwise 0
.TMS320C6400_PLUS	Set to 1 for C6400+, otherwise 0
.TMS320C6700	Set to 1 for C6700 and C6700+, otherwise 0
.TMS320C6700_PLUS	Set to 1 for C6700+, otherwise 0
.LITTLE_ENDIAN	Set to 1 if little-endian mode is selected (the -me assembler option is not used); otherwise 0.
.BIG_ENDIAN	Set to 1 if big-endian mode is selected (the -me assembler option is used); otherwise 0.

**Table 3-3. Assembler Version Symbols**

Symbol name	Description
__TI_ASSEMBLER_VERSION	Defined to a 7-digit integer that takes the 3-digit release version number X.Y.Z and generates an integer XXXYYYZZZ where each portion X, Y and Z is expanded to three digits and concatenated together. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.

### 3.8.6 Substitution Symbols

Symbols can be assigned a string value (variable). This enables you to alias character strings by equating them to symbolic names. Symbols that represent character strings are called substitution symbols. When the assembler encounters a substitution symbol, its string value is substituted for the symbol name. Unlike symbolic constants, substitution symbols can be redefined.

A string can be assigned to a substitution symbol anywhere within a program; for example:

```
.global _table
.asg    "B14", PAGEPTR
.asg    "**+B15(4)", LOCAL1
.asg    "**+B15(8)", LOCAL2

LDW     *+PAGEPTR(_table),A0
NOP     4
STW     A0,LOCAL1
```

When you are using macros, substitution symbols are important because macro parameters are actually substitution symbols that are assigned a macro argument. The following code shows how substitution symbols are used in macros:

```
MAC .macro src1, src2, dst ; Multiply/Accumulate macro
    MPY     src1, src2, src2
    NOP
    ADD     src2, dst, dst
.endm

* MAC macro invocation
MAC     A0,A1,A2
```

For more information about macros, see [Chapter 5, Macro Language](#).

### 3.9 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. The 32-bit ranges of valid expression values are -2147 483 648 to 2147 483 647 for signed values, and 0 to 4 294 967 295 for unsigned values. Three main factors influence the order of expression evaluation:

<b>Parentheses</b>	Expressions enclosed in parentheses are always evaluated first. $8 / (4 / 2) = 4$ , but $8 / 4 / 2 = 1$ You <i>cannot</i> substitute braces ( { } ) or brackets ( [ ] ) for parentheses.
<b>Precedence groups</b>	Operators, listed in <a href="#">Table 3-4</a> , are divided into nine precedence groups. When parentheses do not determine the order of expression evaluation, the highest precedence operation is evaluated first. $8 + 4 / 2 = 10$ ( $4 / 2$ is evaluated first)
<b>Left-to-right evaluation</b>	When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated from left to right, except for Group 1, which is evaluated from right to left. $8 / 4 * 2 = 4$ , but $8 / (4 * 2) = 1$

#### 3.9.1 Operators

[Table 3-4](#) lists the operators that can be used in expressions, according to precedence group.

**Table 3-4. Operators Used in Expressions  
(Precedence)**

Group <sup>(1)</sup>	Operator	Description
1	+	Unary plus
	-	Unary minus
	~	1s complement
	!	Logical NOT
2	*	Multiplication
	/	Division
	%	Modulo
3	+	Addition
	-	Subtraction
4	<<	Shift left
	>>	Shift right
5	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
6	=[=]	Equal to
	!=	Not equal to
7	&	Bitwise AND
8	^	Bitwise exclusive OR (XOR)
9		Bitwise OR

<sup>(1)</sup> **Note:** Group 1 operators are evaluated right to left. All other operators are evaluated left to right.

#### 3.9.2 Expression Overflow and Underflow

The assembler checks for overflow and underflow conditions when arithmetic operations are performed at assembly time. It issues a warning (the message *Value Truncated*) whenever an overflow or underflow occurs. The assembler *does not* check for overflow or underflow in multiplication.

### 3.9.3 Well-Defined Expressions

Some assembler directives require well-defined expressions as operands. Well-defined expressions contain only symbols or assembly-time constants that are defined before they are encountered in the expression. The evaluation of a well-defined expression must be absolute.

This is an example of a well-defined expression:

```
1000h+X
```

where X was previously defined as an absolute symbol.

### 3.9.4 Conditional Expressions

The assembler supports relational operators that can be used in any expression; they are especially useful for conditional assembly. Relational operators include the following:

=	Equal to	!=	Not equal to
<	Less than	<=	Less than or equal to
>	Greater than	>=	Greater than or equal to

Conditional expressions evaluate to 1 if true and 0 if false and can be used only on operands of equivalent types; for example, absolute value compared to absolute value, but not absolute value compared to relocatable value.

### 3.9.5 Legal Expressions

With the exception of the following expression contexts, there is no restriction on combinations of operations, constants, internally defined symbols, and externally defined symbols.

When an expression contains more than one relocatable symbol or cannot be evaluated at assembly time, the assembler encodes a relocation expression in the object file that is later evaluated by the linker. If the final value of the expression is larger in bits than the space reserved for it, you receive an error message from the linker. For more information on relocation expressions, see [Section 2.4](#).

- When using the register relative addressing mode, the expression in brackets or parenthesis must be a well-defined expression, as described in [Section 3.9.3](#). For example:

```
*+A4[15]
```

- Expressions used to describe the offset in register relative addressing mode for the registers B14 and B15, or expressions used as the operand to the branch instruction, are subject to the same limitations. For these two cases, all legal expressions can be reduced to one of two forms:

<i>relocatable symbol</i> ± <i>absolute symbol</i>	B ( <b>extern_1-10</b> )
or	
<i>a well-defined expression</i>	*+B14/B15[ <b>14</b> ]

### 3.9.6 Expression Examples

Following are examples of expressions that use relocatable and absolute symbols. These examples use four symbols that are defined in the same section:

```
.global extern_1 ; Defined in an external module
intern_1: .word 'D' ; Relocatable, defined in
                  ; current module
intern_2          ; Relocatable, defined in
                  ; current module
intern_3          ; Relocatable, defined in
                  ; current module
```

- **Example 1**

In these contexts, there are no limitations on how expressions can be formed.

```
.word    extern_1 * intern_2 - 13    ; Legal
```

```
MVKL    (intern_1 - extern_1),A1    ; Legal
```

- **Example 2**

The first statement in the following example is valid; the statements that follow it are invalid.

```
B (extern_1 - 10)                ; Legal
B (10-extern_1)                  ; Can't negate reloc. symbol
LDW *+B14 (-(intern_1)), A1      ; Can't negate reloc. symbol
LDW *+B14 (extern_1/10), A1      ; / not an additive operator
B (intern_1 + extern_1)          ; Multiple relocatables
```

- **Example 3**

The first statement below is legal; although `intern_1` and `intern_2` are relocatable, their difference is absolute because they are in the same section. Subtracting one relocatable symbol from another reduces the expression to *relocatable symbol + absolute value*. The second statement is illegal because the sum of two relocatable symbols is not an absolute value.

```
B (intern_1 - intern_2 + extern_3)    ; Legal
```

```
B (intern_1 + intern_2 + extern_3)    ; Illegal
```

- **Example 4**

A relocatable symbol's placement in the expression is important to expression evaluation. Although the statement below is similar to the first statement in the previous example, it is illegal because of left-to-right operator precedence; the assembler attempts to add `intern_1` to `extern_3`.

```
B (intern_1 + extern_3 - intern_2)    ; Illegal
```



## 3.10 Source Listings

A source listing shows source statements and the object code they produce. To obtain a listing file, invoke the assembler with the -al (lowercase L) option (see [Section 3.3](#)).

Two banner lines, a blank line, and a title line are at the top of each source listing page. Any title supplied by the .title directive is printed on the title line. A page number is printed to the right of the title. If you do not use the .title directive, the name of the source file is printed. The assembler inserts a blank line below the title line.

Each line in the source file produces at least one line in the listing file. This line shows a source statement number, an SPC value, the object code assembled, and the source statement. [Figure 3-2](#) shows these in an actual listing file.

### Field 1: Source Statement Number

#### Line number

The source statement number is a decimal number. The assembler numbers source lines as it encounters them in the source file; some statements increment the line counter but are not listed. (For example, .title statements and statements following a .nolist are not listed.) The difference between two consecutive source line numbers indicates the number of intervening statements in the source file that are not listed.

#### Include file letter

A letter preceding the line number indicates the line is assembled from the include file designated by the letter.

#### Nesting level number

A number preceding the line number indicates the nesting level of macro expansions or loop blocks.

### Field 2: Section Program Counter

This field contains the SPC value, which is hexadecimal. All sections (.text, .data, .bss, and named sections) maintain separate SPCs. Some directives do not affect the SPC and leave this field blank.

### Field 3: Object Code

This field contains the hexadecimal representation of the object code. All machine instructions and directives use this field to list object code. This field also indicates the relocation type associated with an operand for this line of source code. If more than one operand is relocatable, this column indicates the relocation type for the first operand. The characters that can appear in this column and their associated relocation types are listed below:

!	undefined external reference
'	.text relocatable
+	.sect relocatable
"	.data relocatable
-	.bss, .usect relocatable
%	relocation expression

### Field 4: Source Statement Field

This field contains the characters of the source statement as they were scanned by the assembler. The assembler accepts a maximum line length of 200 characters. Spacing in this field is determined by the spacing in the source statement.

[Figure 3-2](#) shows an assembler listing with each of the four fields identified.

**Figure 3-2. Example Assembler Listing**

Include file letter	Nesting level number	Line number	
	1	1	*****
	2	2	** Global variables
	3	3	*****
	4	4	00000000 .bss var1, 4
	5	5	00000004 .bss var2, 4
	6	6	
	7	7	*****
	8	8	** Include multiply macro
	9	9	*****
	10	10	.copy mpy32.inc
A	1	1	mpy32 .macro A,B
A	2	2	
A	3	3	MPYLH.M1 A,B,A ; tmp1 = A.lo * B.hi
A	4	4	MPYHL.M2 A,B,B ; tmp2 = A.hi * B.lo
A	5	5	
A	6	6	MPYU.M2 A,B,B ; tmp3 = A.lo * B.lo
A	7	7	
A	8	8	ADD.L1 A,B,A ; A = tmp1 + tmp2
A	9	9	
A	10	10	SHL.S1 A,16,A ; A <= 16
A	11	11	
A	12	12	ADD.L1 B,A,A ; A = A + tmp3
A	13	13	.endm
	11	11	
	12	12	*****
	13	13	** _func multiplies 2 global ints
	14	14	*****
	15	15	00000000 .text
	16	16	00000000 _func
	17	17	00000000 0200006C- LDW *+B14(var1),A4
	18	18	00000004 0000016E- LDW *+B14(var2),B0
	19	19	00000008 00006000 NOP 4
	20	20	0000000c mpy32 A4,B0
1	1	1	
1	1	1	0000000c 02009881 MPYLH.M1 A4,B0,A4 ; tmp1 = A.lo * B.hi
1	1	1	00000010 00101882    MPYHL.M2 A4,B0,B0 ; tmp2 = A.hi * B.lo
1	1	1	
1	1	1	00000014 00101F82 MPYU.M2 A4,B0,B0 ; tmp3 = A.lo * B.lo
1	1	1	
1	1	1	00000018 02009078 ADD.L1 A4,B0,A4 ; A = tmp1 + tmp2
1	1	1	
1	1	1	0000001c 02120CA0 SHL.S1 A4,16,A4 ; A <= 16
1	1	1	
1	1	1	00000020 02009078 ADD.L1 B0,A4,A4 ; A = A + tmp3
	21	21	00000024 000C6362 B B3
	22	22	00000028 00008000 NOP 5
	23	23	* end _func
Field 1	Field 2	Field 3	Field 4

### 3.11 Debugging Assembly Source

When you invoke cl6x with --symdebug:dwarf (or -g) when compiling an assembly file, the assembler provides symbolic debugging information that allows you to step through your assembly code in a debugger rather than using the Disassembly window in Code Composer Studio. This enables you to view source comments and other source-code annotations while debugging.

The `.asmfunc` and `.endasmfunc` (see [Mark Function Boundaries](#)) directives enable you to use C characteristics in assembly code that makes the process of debugging an assembly file more closely resemble debugging a C/C++ source file.

The `.asmfunc` and `.endasmfunc` directives allow you to name certain areas of your code, and make these areas appear in the debugger as C functions. Contiguous sections of assembly code that are not enclosed by the `.asmfunc` and `.endasmfunc` directives are automatically placed in assembler-defined functions named with this syntax:

**\$ filename : starting source line : ending source line \$**

If you want to view your variables as a user-defined type in C code, the types must be declared and the variables must be defined in a C file. This C file can then be referenced in assembly code using the `.ref` directive (see [Identify Global Symbols](#)).

#### Example 3-4. Viewing Assembly Variables as C Types C Program

```
typedef struct
{
    int m1;
    int m2;
} X;

X svar = { 1, 2 };
```

#### Example 3-5. Assembly Program for [Example 3-4](#)

```
;-----
; Tell the assembler we're referencing variable "_svar", which is defined in
; another file (cvars.c).
;-----
.ref _svar

;-----
; addfive() - Add five to the second data member of _svar
;-----

.text
.global addfive
addfive: .asmfunc
        LDW    .D2T2    *,B14(_svar+4),B4 ; load svar.m2 into B4
        RET     .S2      B3                ; return from function
        NOP     3         ; delay slots 1-3
        ADD     .D2      5,B4,B4           ; add 5 to B4 (delay slot 4)
        STW     .D2T2    B4,*,B14(_svar+4) ; store B4 back into svar.m2
                                           ; (delay slot 5)
        .endasmfunc
```

[Example 3-4](#) shows the `cvar.c` C program that defines a variable, `svar`, as the structure type `X`. The `svar` variable is then referenced in the `addfive.asm` assembly program in [Example 3-5](#) and 5 is added to `svar`'s second data member.

Compile both source files with the `--symdebug:dwarf` option (`-g`) and link them as follows:

```
cl6x -symdebug:dwarf cvars.c addfive.asm -z -l=lnk.cmd -l=rts6000.lib -o=addfive.out
```

When you load this program into a symbolic debugger, `addfive` appears as a C function. You can monitor the values in `svar` while stepping through `main` just as you would any regular C variable.

### 3.12 Cross-Reference Listings

A cross-reference listing shows symbols and their definitions. To obtain a cross-reference listing, invoke the assembler with the -ax option (see [Section 3.3](#)) or use the .option directive with the X operand (see [Select Listing Options](#)). The assembler appends the cross-reference to the end of the source listing. [Example 3-6](#) shows the four fields contained in the cross-reference listing.

**Example 3-6. An Assembler Cross-Reference Listing**

LABEL	VALUE	DEFN	REF
.BIG_ENDIAN	00000000	0	
.LITTLE_ENDIAN	00000001	0	
.TMS320C6200	00000001	0	
.TMS320C6700	00000000	0	
.TMS320C6X	00000001	0	
_func	00000000'	18	
var1	00000000-	4	17
var2	00000004-	5	18

<b>Label</b>	column contains each symbol that was defined or referenced during the assembly.
<b>Value</b>	column contains an 8-digit hexadecimal number (which is the value assigned to the symbol) or a name that describes the symbol's attributes. A value may also be preceded by a character that describes the symbol's attributes. <a href="#">Table 3-5</a> lists these characters and names.
<b>Definition</b>	(DEFN) column contains the statement number that defines the symbol. This column is blank for undefined symbols.
<b>Reference</b>	(REF) column lists the line numbers of statements that reference the symbol. A blank in this column indicates that the symbol was never used.

**Table 3-5. Symbol Attributes**

Character or Name	Meaning
REF	External reference (global symbol)
UNDF	Undefined
'	Symbol defined in a .text section
"	Symbol defined in a .data section
+	Symbol defined in a .sect section
-	Symbol defined in a .bss or .usect section

**Cross-Reference Listing Not Supported for C6400+**

**Note:** The cross-reference listing capability is not supported for C6400+. You can use the disassembler, the -m linker option or the object file utility (ofd6x) to obtain similar information.

## Assembler Directives

Assembler directives supply data to the program and control the assembly process. Assembler directives enable you to do the following:

- Assemble code and data into specified sections
- Reserve space in memory for uninitialized variables
- Control the appearance of listings
- Initialize memory
- Assemble conditional blocks
- Define global variables
- Specify libraries from which the assembler can obtain macros
- Examine symbolic debugging information

This chapter is divided into two parts: the first part ([Section 4.1](#) through [Section 4.10](#)) describes the directives according to function, and the second part ([Section 4.11](#)) is an alphabetical reference.

Topic	Page
<b>4.1 Directives Summary .....</b>	<b>62</b>
<b>4.2 Directives That Define Sections .....</b>	<b>65</b>
<b>4.3 Directives That Initialize Constants .....</b>	<b>67</b>
<b>4.4 Directives That Perform Alignment and Reserve Space .....</b>	<b>68</b>
<b>4.5 Directives That Format the Output Listings .....</b>	<b>69</b>
<b>4.6 Directives That Reference Other Files .....</b>	<b>70</b>
<b>4.7 Directives That Enable Conditional Assembly .....</b>	<b>71</b>
<b>4.8 Directives That Define Unions or Structures.....</b>	<b>72</b>
<b>4.9 Directives That Define Symbols at Assembly Time.....</b>	<b>74</b>
<b>4.10 Miscellaneous Directives .....</b>	<b>75</b>
<b>4.11 Directives Reference .....</b>	<b>76</b>

## 4.1 Directives Summary

Table 4-1 through Table 4-9 summarize the assembler directives.

Besides the assembler directives documented here, the TMS320C6000™ software tools support the following directives:

- The assembler uses several directives for macros. Macro directives are discussed in [Chapter 5, Macro Language](#); they are not discussed in this chapter.
- The assembly optimizer uses several directives that supply data and control the optimization process. Assembly optimizer directives are discussed in the *TMS320C6000 Optimizing Compiler User's Guide*; they are not discussed in this book.
- The C compiler uses directives for symbolic debugging. Unlike other directives, symbolic debugging directives are not used in most assembly language programs. [Appendix B, Symbolic Debugging Directives](#), discusses these directives; they are not discussed in this chapter.

---

### Labels and Comments Are Not Shown in Syntaxes

**Note:** Any source statement that contains a directive can also contain a label and a comment. Labels begin in the first column (only labels and comments can appear in the first column), and comments must be preceded by a semicolon, or an asterisk if the comment is the only element in the line. To improve readability, labels and comments are not shown as part of the directive syntax.

---

**Table 4-1. Directives That Define Sections**

Mnemonic and Syntax	Description	See
<b>.bss</b> <i>symbol, size in bytes</i> [, <i>alignment</i> [, <i>bank offset</i> ]]	Reserves <i>size</i> bytes in the .bss (uninitialized data) section	<a href="#">.bss topic</a>
<b>.data</b>	Assembles into the .data (initialized data) section	<a href="#">.data topic</a>
<b>.sect</b> " <i>section name</i> "	Assembles into a named (initialized) section	<a href="#">.sect topic</a>
<b>.text</b>	Assembles into the .text (executable code) section	<a href="#">.text topic</a>
<i>symbol</i> <b>.usect</b> " <i>section name</i> ", <i>size in bytes</i> [, <i>alignment</i> [, <i>bank offset</i> ]]	Reserves <i>size</i> bytes in a named (uninitialized) section	<a href="#">.usect topic</a>

**Table 4-2. Directives That Initialize Constants (Data and Memory)**

Mnemonic and Syntax	Description	See
<b>.byte</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more successive bytes in the current section	<a href="#">.byte topic</a>
<b>.char</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more successive bytes in the current section	<a href="#">.char topic</a>
<b>.double</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 64-bit, IEEE double-precision, floating-point constants	<a href="#">.double topic</a>
<b>.field</b> <i>value</i> [, <i>size</i> ]	Initializes a field of <i>size</i> bits (1-32) with <i>value</i>	<a href="#">.field topic</a>
<b>.float</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 32-bit, IEEE single-precision, floating-point constants	<a href="#">.float topic</a>
<b>.half</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 16-bit integers (halfword)	<a href="#">.half topic</a>
<b>.int</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 32-bit integers	<a href="#">.int topic</a>
<b>.long</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 32-bit integers	<a href="#">.long topic</a>
<b>.short</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 16-bit integers (halfword)	<a href="#">.short topic</a>
<b>.string</b> { <i>expr<sub>1</sub></i> " <i>string<sub>1</sub></i> " } [, ... , { <i>expr<sub>n</sub></i> " <i>string<sub>n</sub></i> " }]	Initializes one or more text strings	<a href="#">.string topic</a>
<b>.word</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 32-bit integers	<a href="#">.word topic</a>
<b>.uhalf</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 16-bit integers (halfword)	<a href="#">.uhalf topic</a>
<b>.uint</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 32-bit integers	<a href="#">.uint topic</a>
<b>.ushort</b> <i>value<sub>1</sub></i> [, ... , <i>value<sub>n</sub></i> ]	Initializes one or more 16-bit integers (halfword)	<a href="#">.ushort topic</a>

**Table 4-2. Directives That Initialize Constants (Data and Memory) (continued)**

Mnemonic and Syntax	Description	See
<b>.uword</b> <i>value<sub>1</sub></i> , ... , <i>value<sub>n</sub></i>	Initializes one or more 32-bit integers	<a href="#">.uword topic</a>

**Table 4-3. Directives That Perform Alignment and Reserve Space**

Mnemonic and Syntax	Description	See
<b>.align</b> [ <i>size in bytes</i> ]	Aligns the SPC on a boundary specified by <i>size in bytes</i> , which must be a power of 2; defaults to byte boundary	<a href="#">.align topic</a>
<b>.bes</b> <i>size</i>	Reserves <i>size</i> bytes in the current section; a label points to the end of the reserved space	<a href="#">.bes topic</a>
<b>.space</b> <i>size</i>	Reserves <i>size</i> bytes in the current section; a label points to the beginning of the reserved space	<a href="#">.space topic</a>

**Table 4-4. Directives That Format the Output Listing**

Mnemonic and Syntax	Description	See
<b>.drlst</b>	Enables listing of all directive lines (default)	<a href="#">.drlst topic</a>
<b>.drnolist</b>	Suppresses listing of certain directive lines	<a href="#">.drnolist topic</a>
<b>.fclist</b>	Allows false conditional code block listing (default)	<a href="#">.fclist topic</a>
<b>.fcnolist</b>	Suppresses false conditional code block listing	<a href="#">.fcnolist topic</a>
<b>.length</b> [ <i>page length</i> ]	Sets the page length of the source listing	<a href="#">.length topic</a>
<b>.list</b>	Restarts the source listing	<a href="#">.list topic</a>
<b>.mlist</b>	Allows macro listings and loop blocks (default)	<a href="#">.mlist topic</a>
<b>.mnolist</b>	Suppresses macro listings and loop blocks	<a href="#">.mnolist topic</a>
<b>.nolist</b>	Stops the source listing	<a href="#">.nolist topic</a>
<b>.option</b> <i>option<sub>1</sub></i> , <i>option<sub>2</sub></i> , . . .]	Selects output listing options; available options are A, B, D, H, L, M, N, O, R, T, W, and X	<a href="#">.option topic</a>
<b>.page</b>	Ejects a page in the source listing	<a href="#">.page topic</a>
<b>.sslist</b>	Allows expanded substitution symbol listing	<a href="#">.sslist topic</a>
<b>.ssnolist</b>	Suppresses expanded substitution symbol listing (default)	<a href="#">.ssnolist topic</a>
<b>.tab</b> <i>size</i>	Sets tab to <i>size</i> characters	<a href="#">.tab topic</a>
<b>.title</b> " <i>string</i> "	Prints a title in the listing page heading	<a href="#">.title topic</a>
<b>.width</b> [ <i>page width</i> ]	Sets the page width of the source listing	<a href="#">.width topic</a>

**Table 4-5. Directives That Reference Other Files**

Mnemonic and Syntax	Description	See
<b>.copy</b> [" <i>filename</i> "]	Includes source statements from another file	<a href="#">.copy topic</a>
<b>.def</b> <i>symbol<sub>1</sub></i> , ... , <i>symbol<sub>n</sub></i>	Identifies one or more symbols that are defined in the current module and that can be used in other modules	<a href="#">.def topic</a>
<b>.global</b> <i>symbol<sub>1</sub></i> , ... , <i>symbol<sub>n</sub></i>	Identifies one or more global (external) symbols	<a href="#">.global topic</a>
<b>.include</b> [" <i>filename</i> "]	Includes source statements from another file	<a href="#">.include topic</a>
<b>.mlib</b> [" <i>filename</i> "]	Defines macro library	<a href="#">.mlib topic</a>
<b>.ref</b> <i>symbol<sub>1</sub></i> , ... , <i>symbol<sub>n</sub></i>	Identifies one or more symbols used in the current module that are defined in another module	<a href="#">.ref topic</a>

**Table 4-6. Directives That Enable Conditional Assembly**

Mnemonic and Syntax	Description	See
<b>.break</b> [ <i>well-defined expression</i> ]	Ends .loop assembly if <i>well-defined expression</i> is true. When using the .loop construct, the .break construct is optional.	<a href="#">.break topic</a>
<b>.else</b>	Assembles code block if the .if <i>well-defined expression</i> is false. When using the .if construct, the .else construct is optional.	<a href="#">.else topic</a>

**Table 4-6. Directives That Enable Conditional Assembly (continued)**

Mnemonic and Syntax	Description	See
<b>.elseif</b> <i>well-defined expression</i>	Assembles code block if the <i>.if well-defined expression</i> is false and the <i>.elseif</i> condition is true. When using the <i>.if</i> construct, the <i>.elseif</i> construct is optional.	<a href="#">.elseif topic</a>
<b>.endif</b>	Ends <i>.if</i> code block	<a href="#">.endif topic</a>
<b>.endloop</b>	Ends <i>.loop</i> code block	<a href="#">.endloop topic</a>
<b>.if</b> <i>well-defined expression</i>	Assembles code block if the <i>well-defined expression</i> is true	<a href="#">.if topic</a>
<b>.loop</b> [ <i>well-defined expression</i> ]	Begins repeatable assembly of a code block; the loop count is determined by the <i>well-defined expression</i> .	<a href="#">.loop topic</a>

**Table 4-7. Directives That Define Unions or Structures**

Mnemonic and Syntax	Description	See
<b>.cstruct</b>	Acts like <i>.struct</i> , but adds padding and alignment like that which is done to C structures	<a href="#">.cstruct topic</a>
<b>.cunion</b>	Acts like <i>.union</i> , but adds padding and alignment like that which is done to C structures	<a href="#">.cunion topic</a>
<b>.endstruct</b>	Ends a structure definition	<a href="#">.cstruct/.cunion</a> , <a href="#">.struct</a>
<b>.endunion</b>	Ends a union definition	<a href="#">.cstruct/.cunion</a> , <a href="#">.union</a>
<b>.struct</b>	Begins structure definition	<a href="#">.struct topic</a>
<b>.tag</b>	Assigns structure attributes to a label	<a href="#">.cstruct/.cunion</a> , <a href="#">.struct</a> , <a href="#">.union</a>
<b>.union</b>	Begins a union definition	<a href="#">.union topic</a>

**Table 4-8. Directives That Define Symbols at Assembly Time**

Mnemonic and Syntax	Description	See
<b>.asg</b> [" <i>character string</i> "], <i>substitution symbol</i>	Assigns a character string to <i>substitution symbol</i>	<a href="#">.asg topic</a>
<i>symbol</i> <b>.equ</b> <i>value</i>	Equates <i>value</i> with <i>symbol</i>	<a href="#">.equ topic</a>
<b>.eval</b> <i>well-defined expression</i> , <i>substitution symbol</i>	Performs arithmetic on a numeric <i>substitution symbol</i>	<a href="#">.eval topic</a>
<b>.label</b> <i>symbol</i>	Defines a load-time relocatable label in a section	<a href="#">.label topic</a>
<i>symbol</i> <b>.set</b> <i>value</i>	Equates <i>value</i> with <i>symbol</i>	<a href="#">.set topic</a>
<b>.var</b>	Adds a local substitution symbol to a macro's parameter list	<a href="#">.var topic</a>



**Table 4-9. Directives That Perform Miscellaneous Functions**

Mnemonic and Syntax	Description	See
<b>.asmfunc</b>	Identifies the beginning of a block of code that contains a function	<a href="#">.asmfunc topic</a>
<b>.clink</b> ["section name"]	Enables conditional linking for the current or specified section	<a href="#">.clink topic</a>
<b>.cdecls</b> [options,] "filename"[, "filename2"[, ...]	Share C headers between C and assembly code	<a href="#">.cdecls topic</a>
<b>.emsg</b> string	Sends user-defined error messages to the output device; produces no .obj file	<a href="#">.emsg topic</a>
<b>.end</b>	Ends program	<a href="#">.end topic</a>
<b>.endasmfunc</b>	Identifies the end of a block of code that contains a function	<a href="#">.endasmfunc topic</a>
<b>.mmsg</b> string	Sends user-defined messages to the output device	<a href="#">.mmsg topic</a>
<b>.newblock</b>	Undefines local labels	<a href="#">.newblock topic</a>
<b>.nocmp</b>	Instructs tools to not utilize 16-bit instructions for section	<a href="#">.nocmp topic</a>
<b>.wmsg</b> string	Sends user-defined warning messages to the output device	<a href="#">.wmsg topic</a>

## 4.2 Directives That Define Sections

These directives associate portions of an assembly language program with the appropriate sections:

- The **.bss** directive reserves space in the .bss section for uninitialized variables.
- The **.data** directive identifies portions of code in the .data section. The .data section usually contains initialized data.
- The **.sect** directive defines an initialized named section and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- The **.text** directive identifies portions of code in the .text section. The .text section usually contains executable code.
- The **.usect** directive reserves space in an uninitialized named section. The .usect directive is similar to the .bss directive, but it allows you to reserve space separately from the .bss section.

[Chapter 2, Introduction to Common Object File Format](#), discusses COFF sections in detail.

[Example 4-1](#) shows how you can use sections directives to associate code and data with the proper sections. This is an output listing; column 1 shows line numbers, and column 2 shows the SPC values. (Each section has its own program counter, or SPC.) When code is first placed in a section, its SPC equals 0. When you resume assembling into a section after other code is assembled, the section's SPC resumes counting as if there had been no intervening code.

The directives in [Example 4-1](#) perform the following tasks:

<b>.text</b>	initializes words with the values 1, 2, 3, 4, 5, 6, 7, and 8.
<b>.data</b>	initializes words with the values 9, 10, 11, 12, 13, 14, 15, and 16.
<b>var_defs</b>	initializes words with the values 17 and 18.
<b>.bss</b>	reserves 19 bytes.
<b>xy</b>	reserves 20 bytes.

The .bss and .usect directives do not end the current section or begin new sections; they reserve the specified amount of space, and then the assembler resumes assembling code or data into the current section.

### Example 4-1. Sections Directives

1	*****
2	* Start assembling into the .text section *
3	*****
4	00000000 .text

*Directives That Define Sections***Example 4-1. Sections Directives (continued)**

```

5 00000000 00000001      .word  1,2
   00000004 00000002
6 00000008 00000003      .word  3,4
   0000000c 00000004
7
8      *****
9      *      Start assembling into the .data section      *
10     *****
11 00000000      .data
12 00000000 00000009      .word  9, 10
   00000004 0000000A
13 00000008 0000000B      .word  11, 12
   0000000c 0000000C
14
15     *****
16     *      Start assembling into a named,              *
17     *      initialized section, var_defs                *
18     *****
19 00000000      .sect   "var_defs"
20 00000000 00000011      .word  17, 18
   00000004 00000012
21
22     *****
23     *      Resume assembling into the .data section      *
24     *****
25 00000010      .data
26 00000010 0000000D      .word  13, 14
   00000014 0000000E
27 00000000      .bss    sym, 19    ; Reserve space in .bss
28 00000018 0000000F      .word  15, 16    ; Still in .data
   0000001c 00000010
29
30     *****
31     *      Resume assembling into the .text section      *
32     *****
33 00000010      .text
34 00000010 00000005      .word  5, 6
   00000014 00000006
35 00000000      usym     .usect  "xy", 20    ; Reserve space in xy
36 00000018 00000007      .word  7, 8      ; Still in .text
   0000001c 00000008

```

### 4.3 Directives That Initialize Constants

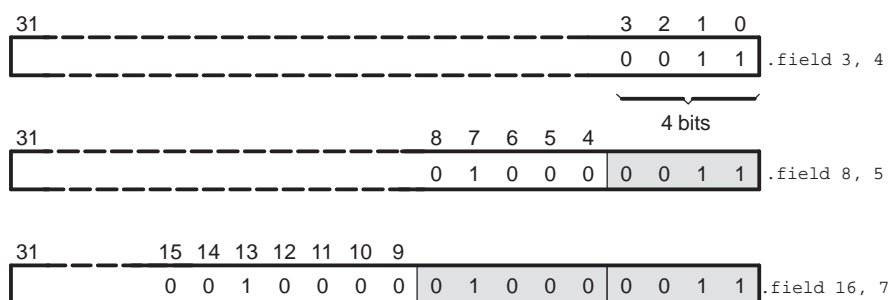
Several directives assemble values for the current section:

- The **.byte** and **.char** directives place one or more 8-bit values into consecutive bytes of the current section. These directives are similar to **.long** and **.word**, except that the width of each value is restricted to eight bits.
- The **.double** directive calculates the double-precision (64-bit) IEEE floating-point representation of one or more floating-point values and stores them in two consecutive words in the current section. The **.double** directive automatically aligns to the double-word boundary.
- The **.field** directive places a single value into a specified number of bits in the current word. With **.field**, you can pack multiple fields into a single word; the assembler does not increment the SPC until a word is filled.

Figure 4-1 shows how fields are packed into a word. Using the following assembled code, notice that the SPC does not change (the fields are packed into the same word):

```
1 00000000 00000003      .field 3,4
2 00000000 00000083      .field 8,5
3 00000000 00002083      .field 16,7
```

Figure 4-1. The **.field** Directive



- The **.float** directive calculates the single-precision (32-bit) IEEE floating-point representation of a single floating-point value and stores it in a word in the current section that is aligned to a word boundary.
- The **.half**, **.uhalf**, **.short**, and **.ushort** directives place one or more 16-bit values into consecutive 16-bit fields (halfwords) in the current section. The **.half** and **.short** directives automatically align to a short (2-byte) boundary.
- The **.int**, **.uint**, **.long**, **.word**, **.uword** directives place one or more 32-bit values into consecutive 32-bit fields (words) in the current section. The **.int**, **.long**, and **.word** directives automatically align to a word boundary.
- The **.string** directive places 8-bit characters from one or more character strings into the current section. This directive is similar to **.byte**, placing an 8-bit character in each consecutive byte of the current section.

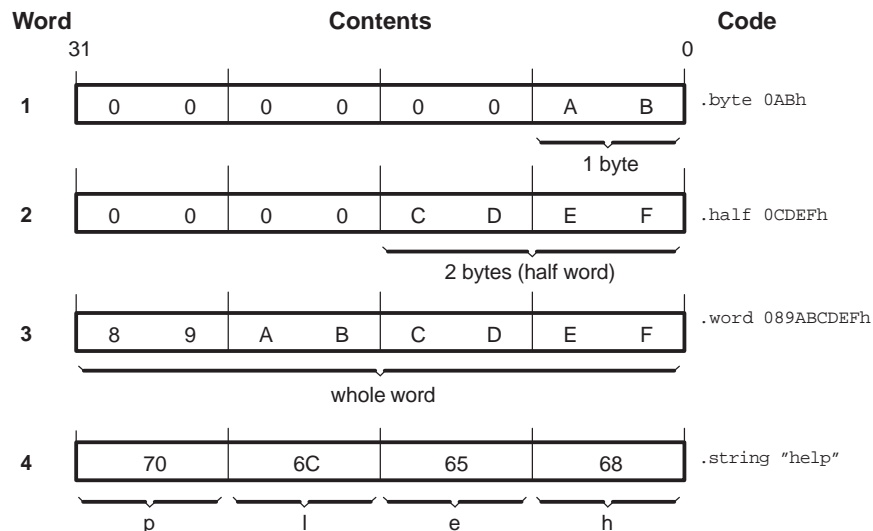
#### Directives That Initialize Constants When Used in a **.struct/endstruct** Sequence

**Note:** The **.byte**, **.char**, **.int**, **.long**, **.word**, **.double**, **.half**, **.short**, **.string**, **.float**, and **.field** directives do not initialize memory when they are part of a **.struct/ .endstruct** sequence; rather, they define a member's size. For more information, see the [.struct/.endstruct directives](#).

Figure 4-2 compares the **.byte**, **.half**, **.word**, and **.string** directives. Using the following assembled code:

```
1 00000000 000000AB      .byte  0ABh
2                               .align 4
3 00000004 0000CDEF      .half  0CDEFh
4 00000008 89ABCDEF      .word   089ABCDEFh
5 0000000c 00000068      .string "help"
   0000000d 00000065
   0000000e 0000006C
   0000000f 00000070
```

Figure 4-2. Initialization Directives



#### 4.4 Directives That Perform Alignment and Reserve Space

These directives align the section program counter (SPC) or reserve space in a section:

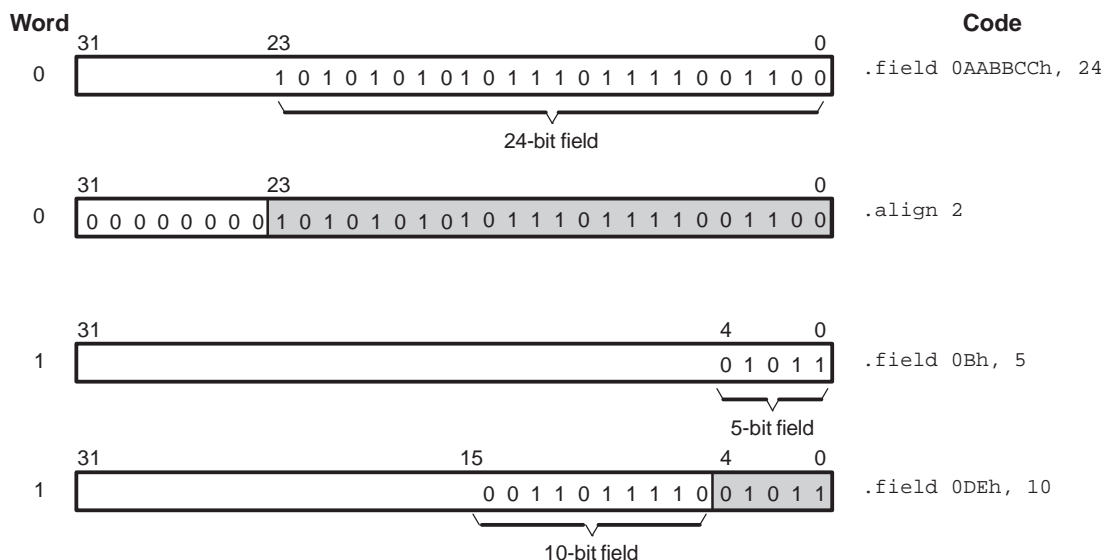
- The **.align** directive aligns the SPC at the next byte boundary. This directive is useful with the **.field** directive when you do not want to pack two adjacent fields in the same byte.

Figure 4-3 demonstrates the **.align** directive. Using the following assembled code:

```

1
2 00000000 00AABBCCh      .field  0AABBCCh,24
3                          .align   2
4 00000000 0BAABBCCh      .field  0Bh,5
5 00000004 000000DE      .field  0DEh,10

```

Figure 4-3. The **.align** Directive

- The **.bes** and **.space** directives reserve a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s.
  - When you use a label with **.space**, it points to the *first* byte that contains reserved bits.
  - When you use a label with **.bes**, it points to the *last* byte that contains reserved bits.

Figure 4-4 shows how the `.space` and `.bes` directives work for the following assembled code:

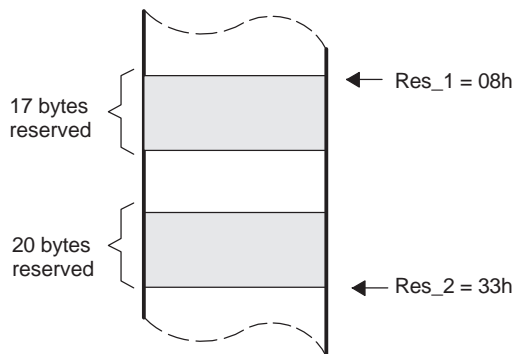
```

1
2 00000000 00000100          .word    100h, 200h
   00000004 00000200
3 00000008          Res_1:   .space    17
4 0000001c 0000000F          .word    15
5 00000033          Res_2:   .bes      20
6 00000034 000000BA          .byte    0BAh

```

Res\_1 points to the first byte in the space reserved by `.space`. Res\_2 points to the last byte in the space reserved by `.bes`.

Figure 4-4. The `.space` and `.bes` Directives



## 4.5 Directives That Format the Output Listings

These directives format the listing file:

- The **`.drlist`** directive causes printing of the directive lines to the listing; the **`.drnolist`** directive turns it off for certain directives. You can use the `.drnolist` directive to suppress the printing of the following directives. You can use the `.drlist` directive to turn the listing on again.

<code>.asg</code>	<code>.eval</code>	<code>.length</code>	<code>.mnolist</code>	<code>.var</code>
<code>.break</code>	<code>.fclist</code>	<code>.mlist</code>	<code>.sslist</code>	<code>.width</code>
<code>.emsg</code>	<code>.fcnolist</code>	<code>.mmsg</code>	<code>.ssnolist</code>	<code>.wmsg</code>

- The source code listing includes false conditional blocks that do not generate code. The **`.fclist`** and **`.fcnolist`** directives turn this listing on and off. You can use the `.fclist` directive to list false conditional blocks exactly as they appear in the source code. You can use the `.fcnolist` directive to list only the conditional blocks that are actually assembled.
- The **`.length`** directive controls the page length of the listing file. You can use this directive to adjust listings for various output devices.
- The **`.list`** and **`.nolist`** directives turn the output listing on and off. You can use the `.nolist` directive to prevent the assembler from printing selected source statements in the listing file. Use the `.list` directive to turn the listing on again.
- The source code listing includes macro expansions and loop blocks. The **`.mlist`** and **`.mnolist`** directives turn this listing on and off. You can use the `.mlist` directive to print all macro expansions and loop blocks to the listing, and the `.mnolist` directive to suppress this listing.
- The **`.option`** directive controls certain features in the listing file. This directive has the following operands:
  - A** turns on listing of all directives and data, and subsequent expansions, macros, and blocks.
  - B** limits the listing of `.byte` and `.char` directives to one line.
  - D** turns off the listing of certain directives (same effect as `.drnolist`).
  - H** limits the listing of `.half` and `.short` directives to one line.

---

*Directives That Reference Other Files*

---

- |          |  |
|----------|--|
| <b>L</b> | limits the listing of <code>.long</code> directives to one line.   |
| <b>M</b> | turns off macro expansions in the listing.   |
| <b>N</b> | turns off listing (performs <code>.nolist</code> ).  |
| <b>O</b> | turns on listing (performs <code>.list</code> ).   |
| <b>R</b> | resets the B, H, L, M, T, and W directives (turns off the limits of B, H, L, M, T, and W).   |
| <b>T</b> | limits the listing of <code>.string</code> directives to one line.   |
| <b>W</b> | limits the listing of <code>.word</code> and <code>.int</code> directives to one line.   |
| <b>X</b> | produces a cross-reference listing of symbols. You can also obtain a cross-reference listing by invoking the assembler with the <code>-x</code> option (see <a href="#">Section 3.3</a> ). |
- The **`.page`** directive causes a page eject in the output listing.
  - The source code listing includes substitution symbol expansions. The **`.sslist`** and **`.ssnolist`** directives turn this listing on and off. You can use the `.sslist` directive to print all substitution symbol expansions to the listing, and the `.ssnolist` directive to suppress this listing. These directives are useful for debugging the expansion of substitution symbols.
  - The **`.tab`** directive defines tab size.
  - The **`.title`** directive supplies a title that the assembler prints at the top of each page.
  - The **`.width`** directive controls the page width of the listing file. You can use this directive to adjust listings for various output devices.

## 4.6 Directives That Reference Other Files

These directives supply information for or about other files that can be used in the assembly of the current file:

- The **`.copy`** and **`.include`** directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from a copied file are printed in the listing file; the statements read from an included file are *not* printed in the listing file.
- The **`.def`** directive identifies a symbol that is defined in the current module and that can be used in another module. The assembler includes the symbol in the symbol table.
- The **`.global`** directive declares a symbol external so that it is available to other modules at link time. (For more information about global symbols, see [Section 2.7.1](#), *External Symbols*). The `.global` directive does double duty, acting as a `.def` for defined symbols and as a `.ref` for undefined symbols. The linker resolves an undefined global symbol reference only if the symbol is used in the program. The `.global` directive declares a 16-bit symbol.
- The **`.mlib`** directive supplies the assembler with the name of an archive library that contains macro definitions. When the assembler encounters a macro that is not defined in the current module, it searches for it in the macro library specified with `.mlib`.
- The **`.ref`** directive identifies a symbol that is used in the current module but is defined in another module. The assembler marks the symbol as an undefined external symbol and enters it in the object symbol table so the linker can resolve its definition. The `.ref` directive forces the linker to resolve a symbol reference.

## 4.7 Directives That Enable Conditional Assembly

Conditional assembly directives enable you to instruct the assembler to assemble certain sections of code according to a true or false evaluation of an expression. Two sets of directives allow you to assemble conditional blocks of code:

- The **.if/.elseif/.else/.endif** directives tell the assembler to conditionally assemble a block of code according to the evaluation of an expression.

<b>.if</b> <i>well-defined expression</i>	marks the beginning of a conditional block and assembles code if the <i>.if well-defined expression</i> is true.
<b>[.elseif</b> <i>well-defined expression</i> ]	marks a block of code to be assembled if the <i>.if well-defined expression</i> is false and the <i>.elseif</i> condition is true.
<b>.else</b>	marks a block of code to be assembled if the <i>.if well-defined expression</i> is false and any <i>.elseif</i> conditions are false.
<b>.endif</b>	marks the end of a conditional block and terminates the block.

- The **.loop/.break/.endloop** directives tell the assembler to repeatedly assemble a block of code according to the evaluation of an expression.

<b>.loop</b> [ <i>well-defined expression</i> ]	marks the beginning of a repeatable block of code. The optional expression evaluates to the loop count.
<b>.break</b> [ <i>well-defined expression</i> ]	tells the assembler to assemble repeatedly when the <i>.break well-defined expression</i> is false and to go to the code immediately after <i>.endloop</i> when the expression is true or omitted.
<b>.endloop</b>	marks the end of a repeatable block.

The assembler supports several relational operators that are useful for conditional expressions. For more information about relational operators, see [Section 3.9.4, Conditional Expressions](#).

## **4.8 Directives That Define Unions or Structures**

These directives set up C or C-like structures or unions in assembly code.



- The **.cstruct/.endstruct** directives set up C structure definitions. The **.cunion/.endunion** directives set up C-like union definitions. The **.tag** directive assigns the C structure or union characteristics to a label. The **.cstruct/.endstruct** directives allow you to organize your information into structures so that similar elements can be grouped together. Similarly, the **.cunion/.endunion** directives allow you to organize your information into unions. Element offset calculation is left up to the assembler. These directives do not allocate memory. They simply create a symbolic template that can be used repeatedly. The **.cstruct** and **.cunion** directives force the same alignment and padding as used by the C compiler when such types are nested within compound data structures..

The **.tag** directive assigns a label to a structure. This simplifies the symbolic representation and also provides the ability to define structures that contain other structures. The **.tag** directive does not allocate memory, and the structure tag (stag) must be defined before it is used.

- The **.struct/.endstruct** directives set up C-like structure definitions. The **.union/.endunion** directives set up C-like union definitions. The **.tag** directive assigns the C-like structure or union characteristics to a label.

The **.struct/.endstruct** directives allow you to organize your information into structures so that similar elements can be grouped together. Similarly, the **.union/.endunion** directives allow you to organize your information into unions. Element offset calculation is left up to the assembler. These directives do not allocate memory. They simply create a symbolic template that can be used repeatedly.

The **.tag** directive assigns a label to a structure or union. This simplifies the symbolic representation and also provides the ability to define structures that contain other structures. The **.tag** directive does not allocate memory, and the structure tag (stag) must be defined before it is used.

```
COORDT .struct                ; structure tag definition
X      .byte                  ;
Y      .byte
T_LEN  .endstruct

COORD  .tag COORDT            ; declare COORD (coordinate)
      .bss COORD, T_LEN      ; actual memory allocation

LDB    *+B14(COORD.Y), A2 ; move member Y of structure
                        ; COORD into register A2
```

## 4.9 Directives That Define Symbols at Assembly Time

Assembly-time symbol directives equate meaningful symbol names to constant values or strings.

- The **.asg** directive assigns a character string to a substitution symbol. The value is stored in the substitution symbol table. When the assembler encounters a substitution symbol, it replaces the symbol with its character string value. Substitution symbols can be redefined.

```
.asg "10, 20, 30, 40", coefficients
.byte coefficients
```

- The **.eval** directive evaluates a well-defined expression, translates the results into a character string, and assigns the character string to a substitution symbol. This directive is most useful for manipulating counters:

```
.asg      1, x
.loop
.byte     x*10h
.break    x = 4
.eval     x+1, x
.endloop
```

- The **.label** directive defines a special symbol that refers to the load-time address within the current section. This is useful when a section loads at one address but runs at a different address. For example, you may want to load a block of performance-critical code into slower off-chip memory to save space and move the code to high-speed on-chip memory to run. See the [.label topic](#) for an example using a load-time address label.
- The **.set** and **.equ** directives set a constant value to a symbol. The symbol is stored in the symbol table and cannot be redefined; for example:

```
bval .set 1000h
     .long bval, bval*2, bval+12
     MVK   bval, A2
```

The **.set** and **.equ** directives produce no object code. The two directives are identical and can be used interchangeably.

## 4.10 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler -gw option to generate debug information for separate functions.
- The **.cdecls** directive enables programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between C and assembly code.
- The **.clink** directive sets the STYP\_CLINK flag in the type field for the named section. The .clink directive can be applied to initialized or uninitialized sections. The STYP\_CLINK flag enables conditional linking by telling the linker to leave the section out of the final COFF output of the linker if there are no references found to any symbol in the section.
- The **.end** directive terminates assembly. If you use the .end directive, it should be the last source statement of a program. This directive has the same effect as an end-of-file character.
- The **.newblock** directive resets local labels. Local labels are symbols of the form \$n, where n is a decimal digit, or of the form NAME?, where you specify NAME. They are defined when they appear in the label field. Local labels are temporary labels that can be used as operands for jump instructions. The .newblock directive limits the scope of local labels by resetting them after they are used. For more information, see [Section 3.8.2, Local Labels](#).
- The **.nocmp** directive for C6400+ instructs the tools to not utilize 16-bit instructions for the section .nocmp appears in.

These three directives enable you to define your own error and warning messages:

- The **.emsg** directive sends error messages to the standard output device. The .emsg directive generates errors in the same manner as the assembler, incrementing the error count and preventing the assembler from producing an object file.
- The **.mmsg** directive sends assembly-time messages to the standard output device. The .mmsg directive functions in the same manner as the .emsg and .wmsg directives but does not set the error count or the warning count. It does not affect the creation of the object file.
- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same manner as the .emsg directive but increments the warning count rather than the error count. It does not affect the creation of the object file.

For more information about using the error and warning directives in macros, see [Section 5.7, Producing Messages in Macros](#).

## 4.11 Directives Reference

The remainder of this chapter is a reference. Generally, the directives are organized alphabetically, one directive per topic. Related directives (such as `.if/.else/.endif`), however, are presented together in one topic.

### **.align**

#### ***Align SPC on the Next Boundary***

##### **Syntax**

**.align** [*size in bytes*]

##### **Description**

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size in bytes* parameter. The *size* can be any power of 2, although only certain values are useful for alignment. An operand of 1 aligns the SPC on the next byte boundary, and this is the default if no *size in bytes* is given. The assembler assembles words containing null values (0) up to the next *size in bytes* boundary:

1	aligns SPC to byte boundary
2	aligns SPC to halfword boundary
4	aligns SPC to word boundary
8	aligns SPC to doubleword boundary
128	aligns SPC to page boundary

Using the `.align` directive has two effects:

- The assembler aligns the SPC on an x-byte boundary *within* the current section.
- The assembler sets a flag that forces the linker to align the section so that individual alignments remain intact when a section is loaded into memory.

##### **Example**

This example shows several types of alignment, including `.align 2`, `.align 8`, and a default `.align`.

```

1 00000000 00000004      .byte      4
2                          .align 2
3 00000002 00000045      .string    "Errorcnt"
  00000003 00000072
  00000004 00000072
  00000005 0000006F
  00000006 00000072
  00000007 00000063
  00000008 0000006E
  00000009 00000074
4
5 00000008 0003746E      .align
6 00000008 002B746E      .field      3,3
7                          .field      5,4
8 0000000c 00000003      .align      2
9                          .field      3,3
10 00000010 00000005     .align      8
11                          .field      5,4
12 00000011 00000004     .align
                          .field      4

```

## **.asg**

### **Assign a Substitution Symbol**

#### **Syntax**

**.asg** ["*character string*"], *substitution symbol*

**.eval** *well-defined expression*, *substitution symbol*

#### **Description**

The **.asg** directive assigns character strings to substitution symbols. Substitution symbols are stored in the substitution symbol table. The **.asg** directive can be used in many of the same ways as the **.set** directive, but while **.set** assigns a constant value (which cannot be redefined) to a symbol, **.asg** assigns a character string (which can be redefined) to a substitution symbol.

- The assembler assigns the *character string* to the substitution symbol. The quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the substitution symbol.
- The *substitution symbol* must be a valid symbol name. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (\_), and the dollar sign (\$).

The **.eval** directive performs arithmetic on substitution symbols, which are stored in the substitution symbol table. This directive evaluates the *well-defined expression* and assigns the string value of the result to the substitution symbol. The **.eval** directive is especially useful as a counter in **.loop**/**.endloop** blocks.

- The *well-defined expression* is an alphanumeric expression in which all symbols have been previously defined in the current source module, so that the result is an absolute.
- The *substitution symbol* must be a valid symbol name. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (\_), and the dollar sign (\$).

#### **Example**

This example shows how **.asg** and **.eval** can be used.

```

1                                     .sslist ; show expanded substitution symbols
2
3                                     .asg  **B14(100), GLOB100
4                                     .asg  **B15(4), ARG0
5
6 00000000 003B22E4                LDW    GLOB100,A0
#                                     LDW    **B14(100),A0
7 00000004 00BC22E4                LDW    ARG0,A1
#                                     LDW    **B15(4),A1
8 00000008 00006000                NOP    4
9 0000000c 010401E0                ADD    A0,A1,A2
10
11                                     .asg  0,x
12                                     .loop  5
13                                     .word  100*x
14                                     .eval  x+1,x
15                                     .endloop
1 00000010 00000000                .word  100*x
#                                     .word  100*0
1                                     .eval  x+1,x
#                                     .eval  0+1,x
1 00000014 00000064                .word  100*x
#                                     .word  100*1
1                                     .eval  x+1,x
#                                     .eval  1+1,x
1 00000018 000000C8                .word  100*x
#                                     .word  100*2
1                                     .eval  x+1,x
#                                     .eval  2+1,x

```

## Directives Reference

```

1      0000001c 0000012C      .word  100*x
#
1      .eval  x+1,x
#      .eval  3+1,x
1      00000020 00000190      .word  100*x
#      .word  100*4
1      .eval  x+1,x
#      .eval  4+1,x

```

### **.asmfunc/.endasm**    *Mark Function Boundaries*

**Syntax**                    *symbol* **.asmfunc**

**.endasm**

**Description**

The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler -g option (--symdebug:dwarf) to allow sections assembly code to be debugged in the same manner as C/C++ functions.

You should not use the same directives generated by the compiler (see [Appendix B](#)) to accomplish assembly debugging; those directives should be used only by the compiler to generate symbolic debugging information for C/C++ source files.

The **.asmfunc** and **.endasmfunc** directives cannot be used when invoking the compiler with the backwards-compatibility --symdebug:coff option. This option instructs the compiler to use the obsolete COFF symbolic debugging format, which does not support these directives.

The *symbol* is a label that must appear in the label field.

Consecutive ranges of assembly code that are not enclosed within a pair of **.asmfunc** and **.endasmfunc** directives are given a default name in the following format:

**\$ filename : beginning source line : ending source line \$**

**Example**

In this example the assembly source generates debug information for the user\_func section.

```

1 00000000      .sect  ".text"
2
3      .global userfunc
4      .global _printf
5
6      userfunc: .asmfunc
7 00000000 00000010!      CALL    .S1    _printf
8 00000004 01BC94F6      STW     .D2T2  B3,*B15--(16)
9 00000008 01800E2A'      MVKL    .S2    RL0,B3
10 0000000c 01800028+      MVKL    .S1    SL1+0,A3
11 00000010 01800068+      MVKH    .S1    SL1+0,A3
12 00000014 01BC22F5      STW     .D2T1  A3,*+B15(4)
13 00000018 0180006A' ||  MVKH    .S2    RL0,B3
14
15 0000001c 01BC92E6  RL0:  LDW     .D2T2  *++B15(16),B3
16 00000020 020008C0      ZERO    .D1    A4
17 00000024 00004000      NOP
18 00000028 000C0362      RET     .S2    B3
19 0000002c 00008000      NOP
20      .endasmfunc
21
22 00000000      .sect  ".const"
23 00000000 00000048  SL1:  .string "Hello World!",10,0
    00000001 00000065
    00000002 0000006C
    00000003 0000006C
    00000004 0000006F
    00000005 00000020
    00000006 00000057
    00000007 0000006F

```

```

00000008 00000072
00000009 0000006C
0000000a 00000064
0000000b 00000021
0000000c 0000000A
0000000d 00000000

```

## **.bss** *Reserve Space in the .bss Section*

**Syntax** `.bss symbol, size in bytes[, alignment[, bank offset]]`

**Description**

The **.bss** directive reserves space for variables in the .bss section. This directive is usually used to allocate space in RAM.

- The *symbol* is a required parameter. It defines a label that points to the first location reserved by the directive. The symbol name must correspond to the variable that you are reserving space for.
- The *size in bytes* is a required parameter; it must be an absolute expression. The assembler allocates size bytes in the .bss section.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. This boundary indicates the size of the slot in bytes and must be set to a power of 2. If the SPC is aligned to the specified boundary, it is not incremented.
- The *bank offset* is an optional parameter that ensures that the space allocated to the symbol occurs on a specific memory bank boundary. The bank offset value measures the number of bytes to offset from the alignment specified before assigning the symbol to that location.

For more information about COFF sections, see [Chapter 2, Introduction to Common Object File Format](#).

**Example**

In this example, the .bss directive is used to allocate space for a variable, array. The symbol array points to 100 bytes of uninitialized space (at .bss SPC = 0). Symbols declared with the .bss directive can be referenced in the same manner as other symbols and can also be declared global.

```

1          *****
2          ** Start assembling into .text section. **
3          *****
4 00000000          .text
5 00000000 008001A0      MV      A0,A1
6
7          *****
8          ** Allocate 100 bytes in .bss.          **
9          *****
10 00000000          .bss array,100
11
12          *****
13          ** Still in .text                        **
14          *****
15 00000004 010401A0      MV      A1,A2
16
17          *****
18          ** Declare external .bss symbol          **
19          *****
20          .global array

```

**.byte/.char****Initialize Byte**

---

**Syntax****.byte** *value*<sub>1</sub>[, ... , *value*<sub>*n*</sub>]**.char** *value*<sub>1</sub>[, ... , *value*<sub>*n*</sub>]**Description**

The **.byte** and **.char** directives place one or more values into consecutive bytes of the current section. A *value* can be one of the following:

- An expression that the assembler evaluates and treats as an 8-bit signed number
- A character string enclosed in double quotes. Each character in a string represents a separate value, and values are stored in consecutive bytes. The entire string *must* be enclosed in quotes.

The first byte occupies the eight least significant bits of a full 32-bit word. The second byte occupies bits eight through 15 while the third byte occupies bits 16 through 23. The assembler truncates values greater than eight bits. You can use up to 100 value parameters, but the total line length cannot exceed 200 characters.

If you use a label, it points to the location of the first byte that is initialized.

When you use **.byte** or **.char** in a **.struct/.endstruct** sequence, **.byte** and **.char** define a member's size; they do not initialize memory. For more information, see the [.struct/.endstruct/.tag](#) topic .

**Example**

In this example, 8-bit values (10, -1, abc, and a) are placed into consecutive bytes in memory with **.byte** and **.char**. The label **strx** has the value 0h, which is the location of the first initialized byte. The label **stry** has the value 6h, which is the first byte initialized by the **.char** directive.

```

1 00000000 0000000A strx   .byte   10,-1,"abc",'a'
   00000001 000000FF
   00000002 00000061
   00000003 00000062
   00000004 00000063
   00000005 00000061
2 00000006 00000008 stry   .char   8,-3,"def",'b'
```



---

**.cdecls**                      **Share C Headers Between C and Assembly Code**


---

**Syntax**                      **Single Line:**

```
.cdecls [options,] "filename" [, "filename2" [...]]
```

**Syntax**                      **Multiple Lines:**

```
.cdecls [options]
```

```
%{
```

```
/*-----*/
```

```
/* C/C++ code - Typically a list of #includes and a few defines */
```

```
/*-----*/
```

```
%}
```

**Description**

The **.cdecls** directive allows programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between the C and assembly code. Any legal C/C++ can be used in a **.cdecls** block and the C/C++ declarations cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code; such as calling functions, allocating space, and accessing structure members; using the equivalent assembly mechanisms. While function and variable definitions are ignored, most common C/C++ elements are converted to assembly, for instance: enumerations, (non-function-like) macros, function and variable prototypes, structures, and unions.

The **.cdecls** options control whether the code is treated as C or C++ code; and how the **.cdecls** block and converted code are presented. Options must be separated by commas; they can appear in any order:

- |               |   |
|---------------|---|
| <b>C</b>      | Treat the code in the <b>.cdecls</b> block as C source code (default).  |
| <b>CPP</b>    | Treat the code in the <b>.cdecls</b> block as C++ source code. This is the opposite of the C option.  |
| <b>NOLIST</b> | Do not include the converted assembly code in any listing file generated for the containing assembly file (default).  |
| <b>LIST</b>   | Include the converted assembly code in any listing file generated for the containing assembly file. This is the opposite of the NOLIST option.                        |
| <b>NOWARN</b> | Do not emit warnings on STDERR about C/C++ constructs that cannot be converted while parsing the <b>.cdecls</b> source block (default).                               |
| <b>WARN</b>   | Generate warnings on STDERR about C/C++ constructs that cannot be converted while parsing the <b>.cdecls</b> source block. This is the opposite of the NOWARN option. |

In the single-line format, the options are followed by one or more filenames to include. The filenames and options are separated by commas. Each file listed acts as if **#include "filename"** was specified in the multiple-line format.

In the multiple-line format, the line following **.cdecls** must contain the opening **.cdecls** block indicator **%{**. Everything after the **%{**, up to the closing block indicator **%}**, is treated as C/C++ source and processed. Ordinary assembler processing then resumes on the line following the closing **%}**.

The text within **%{** and **%}** is passed to the C/C++ compiler to be converted into assembly language. Much of C language syntax, including function and variable definitions as well as function-like macros, is not supported and is ignored during the conversion. However, all of what traditionally appears in C header files is supported, including function and variable prototypes; structure and union declarations; non-function-like macros; enumerations; and **#define**'s. The resulting assembly language

is included in the assembly file at the point of the `.cdecls` directive. If the `LIST` option is used, the converted assembly statements are printed in the listing file.

The assembly resulting from the `.cdecls` directive is treated similarly to a `.include` file. Therefore the `.cdecls` directive can be nested within a file being copied or included. The assembler limits nesting to ten levels; the host operating system may set additional restrictions. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. An A indicates the first copied file, B indicates a second copied file, etc.

The `.cdecls` directive can appear anywhere in an assembly source file, and can occur multiple times within a file. However, the C/C++ environment created by one `.cdecls` is **not** inherited by a later `.cdecls`; the C/C++ environment starts new for each `.cdecls`.

See [Chapter 12, Sharing C Header Files in Assembly Source](#), for more information on setting up and using `.cdecls` with C header files.

### Example

In this example, the `.cdecls` directive is used call the C header.h file.

#### C Header File:

```
#define WANT_ID 10
#define NAME "John\n"

extern int a_variable;
extern float cvt_integer(int src);

struct myCstruct { int member_a; float member_b; };

enum status_enum { OK = 1, FAILED = 256, RUNNING = 0 };
```

#### Source File:

`.cdecls C,LIST,"myheader.h"`

```
size:      .int $sizeof(myCstruct)
aoffset:   .int myCstruct.member_a
boffset:   .int myCstruct.member_b
okvalue:   .int status_enum.OK
failval:   .int status_enum.FAILED
           .if $defined(WANT_ID)
id         .cstring NAME
           .endif
```

#### Listing File:

```

1                      .cdecls C,LIST,"myheader.h"
A 1                      ; -----
A 2                      ; Assembly Generated from C/C++ Source Code
A 3                      ; -----
A 4
A 5                      ; ===== MACRO DEFINITIONS =====
A 6                      .define "10",WANT_ID
A 7                      .define "" "John\n",NAME
A 8
A 9                      ; ===== TYPE DEFINITIONS =====
A 10                     status_enum      .enum
A 11                     00000001 OK        .emember 1
A 12                     00000100 FAILED   .emember 256
A 13                     00000000 RUNNING  .emember 0
A 14                     .endenum
A 15
A 16                     myCstruct         .struct 0,4          ; struct size=(8 bytes|64 bits),
alignment=4
A 17                     00000000 member_a .field 32           ; int member_a - offset 0 bytes, size
(4 bytes|32 bits)
A 18                     00000004 member_b .field 32           ; float member_b - offset 4 bytes, size
```

---

**.clink** — *Conditionally Leave Section Out of COFF Output*


---

```
(4 bytes|32 bits)
A 19          00000008          .endstruct          ; final size=(8 bytes|64 bits)
A 20
A 21          ; ===== EXTERNAL FUNCTIONS =====
A 22          .global _cvt_integer
A 23
A 24          ; ===== EXTERNAL VARIABLES =====
A 25          .global _a_variable
      2 00000000 00000008 size:  .int $sizeof(myCstruct)
      3 00000004 00000000 aoffset: .int myCstruct.member_a
      4 00000008 00000004 boffset: .int myCstruct.member_b
      5 0000000c 00000001 okvalue: .int status_enum.OK
      6 00000010 00000100 failval: .int status_enum.FAILED
      7          .if $defined(WANT_ID)
      8 00000014 0000004A id      .cstring NAME
      00000015 0000006F
      00000016 00000068
      00000017 0000006E
      00000018 0000000A
      00000019 00000000
      9          .endif
```

---

**.clink** *Conditionally Leave Section Out of COFF Output*


---

**Syntax** `.clink ["section name"]`

**Description** The **.clink** directive sets up conditional linking for a section by setting the STYP\_CLINK flag in the type field for *section name*. The **.clink** directive can be applied to initialized or uninitialized sections.

The *section name* identifies the section. If **.clink** is used without a section name, it applies to the current initialized section. If **.clink** is applied to an uninitialized section, the section name is required. The section name is significant to 200 characters and must be enclosed in double quotes. A section name can contain a subsection name in the form *section name:subsection name*.

The **.clink** directive tells the linker to leave the section out of the final COFF output of the linker if there are no references found in a linked section to any symbol defined in the specified section. The **-a** linker option produces the final COFF output in the form of an absolute, executable output module.

A section in which the entry point of a C program is defined cannot be marked as a conditionally linked section.

**Example** In this example, the Vars and Counts sections are set for conditional linking.

```
1 00000000          .sect "Vars"
2          .clink
3          ; Vars section is conditionally linked
4
5 00000000 0000001A X:      .word 01Ah
6 00000004 0000001A Y:      .word 01Ah
7 00000008 0000001A Z:      .word 01Ah
8 00000000          .sect "Counts"
9          .clink
10         ; Counts section is conditionally linked
11
12 00000000 0000001A XCount: .word 01Ah
13 00000004 0000001A YCount: .word 01Ah
14 00000008 0000001A ZCount: .word 01Ah
15 00000000          .text
16         ; By default, .text is unconditionally linked
17
18 00000000 00B802C4      LDH    *B14,A1
19 00000004 00000028+    MVKL    X,A0
20 00000008 00000068+    MVKH    X,A0
21         ; These references to symbol X cause the Vars
```

**.clink** — *Conditionally Leave Section Out of COFF Output*

---

```
22                                ; section to be linked into the COFF output
23 0000000c 00040AF8            CMPLT  A0,A1,A0
```

## .copy/.include

### Copy Source File

#### Syntax

```
.copy ["filename"]
```

```
.include ["filename"]
```

#### Description

The **.copy** and **.include** directives tell the assembler to read source statements from a different file. The statements that are assembled from a copy file are printed in the assembly listing. The statements that are assembled from an included file are *not* printed in the assembly listing, regardless of the number of **.list**/**.nolist** directives assembled.

When a **.copy** or **.include** directive is assembled, the assembler:

1. Stops assembling statements in the current source file
2. Assembles the statements in the copied/included file
3. Resumes assembling statements in the main source file, starting with the statement that follows the **.copy** or **.include** directive

The *filename* is a required parameter that names a source file. It can be enclosed in double quotes and must follow operating system conventions. If *filename* starts with a number the double quotes are required.

You can specify a full pathname (for example, /320tools/file1.asm). If you do not specify a full pathname, the assembler searches for the file in:

1. The directory that contains the current source file
2. Any directories named with the **-I** assembler option
3. Any directories specified by the **C6X\_A\_DIR** or **A\_DIR** environment variable

For more information about the **-I** option, **C6x\_A\_DIR**, and **A\_DIR**, see [Section 3.4, Naming Alternate Directories for Assembler Input](#).

The **.copy** and **.include** directives can be nested within a file being copied or included. The assembler limits nesting to 32 levels; the host operating system may set additional restrictions. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. A indicates the first copied file, B indicates a second copied file, etc.

#### Example 1

In this example, the **.copy** directive is used to read and assemble source statements from other files; then, the assembler resumes assembling into the current file.

The original file, **copy.asm**, contains a **.copy** statement copying the file **byte.asm**. When **copy.asm** assembles, the assembler copies **byte.asm** into its place in the listing (note listing below). The copy file **byte.asm** contains a **.copy** statement for a second file, **word.asm**.

When it encounters the **.copy** statement for **word.asm**, the assembler switches to **word.asm** to continue copying and assembling. Then the assembler returns to its place in **byte.asm** to continue copying and assembling. After completing assembly of **byte.asm**, the assembler returns to **copy.asm** to assemble its remaining statement.

copy.asm (source file)	byte.asm (first copy file)	word.asm (second copy file)
<pre>.space 29 .copy "byte.asm" ** Back in original file .string "done"</pre>	<pre>** In byte.asm .byte 32, 1+ 'A' .copy "word.asm" ** Back in byte.asm .byte 67h + 3q</pre>	<pre>** In word.asm .word 0ABCDh, 56q</pre>

**.copy/.include — Copy Source File****Listing file:**

```

1 00000000          .space 29
2          .copy "byte.asm"
A 1          ** In byte.asm
A 2 0000001d 00000020  .byte 32,1+ 'A'
   0000001e 00000042
A 3          .copy "word.asm"
B 1          ** In word.asm
B 2 00000020 0000ABCD  .word 0ABCDh, 56q
   00000024 0000002E
A 4          ** Back in byte.asm
A 5 00000028 0000006A  .byte 67h + 3q
   3
   4          ** Back in original file
5 00000029 00000064  .string "done"
   0000002a 0000006F
   0000002b 0000006E
   0000002c 00000065

```

**Example 2**

In this example, the `.include` directive is used to read and assemble source statements from other files; then, the assembler resumes assembling into the current file. The mechanism is similar to the `.copy` directive, except that statements are not printed in the listing file.

<b>include.asm (source file)</b>	<b>byte2.asm (first copy file)</b>	<b>word2.asm (second copy file)</b>
<pre> .space 29 .include "byte2.asm" ** Back in original file .string "done" </pre>	<pre> ** In byte2.asm .byte 32,1+ 'A' .include "word2.asm" ** Back in byte2.asm .byte 67h + 3q </pre>	<pre> ** In word2.asm .word 0ABCDh, 56q </pre>

**Listing file:**

```

1 00000000          .space 29
2          .include "byte2.asm"
3
4          ** Back in original file
5 00000029 00000064  .string "done"
   0000002a 0000006F
   0000002b 0000006E
   0000002c 00000065

```

**.cstruct/.cunion/.endstruct/.endunion/.tag**
**Declare C Structure Type**

Syntax	[stag]	<b>.cstruct .cunion</b>	[expr]
	[mem <sub>0</sub> ]	<i>element</i>	[expr <sub>0</sub> ]
	[mem <sub>1</sub> ]	<i>element</i>	[expr <sub>1</sub> ]
	.	.	.
	.	.	.
	.	.	.
	[mem <sub>n</sub> ]	<b>.tag stag</b>	[expr <sub>n</sub> ]
	[mem <sub>N</sub> ]	<i>element</i>	[expr <sub>N</sub> ]
	[size]	<b>.endstruct .endunion</b>	
	<i>label</i>	<b>.tag</b>	<i>stag</i>
Description	<p>The <b>.cstruct</b> and <b>.cunion</b> directives have been added to support ease of sharing of common data structures between assembly and C code. The <b>.cstruct</b> and <b>.cunion</b> directives can be used exactly like the existing <b>.struct</b> and <b>.union</b> directives except that they are guaranteed to perform data layout matching the layout used by the C compiler for C struct and union data types.</p> <p>In particular, the <b>.cstruct</b> and <b>.cunion</b> directives force the same alignment and padding as used by the C compiler when such types are nested within compound data structures.</p> <p>The <b>.endstruct</b> directive terminates the structure definition. The <b>.endunion</b> directive terminates the union definition.</p> <p>The <b>.tag</b> directive gives structure characteristics to a <i>label</i>, simplifying the symbolic representation and providing the ability to define structures that contain other structures. The <b>.tag</b> directive does not allocate memory. The structure tag (<i>stag</i>) of a <b>.tag</b> directive must have been previously defined.</p> <p>Following are descriptions of the parameters used with the <b>.struct</b>, <b>.endstruct</b>, and <b>.tag</b> directives:</p> <ul style="list-style-type: none"> <li>• The <i>stag</i> is the structure's tag. Its value is associated with the beginning of the structure. If no <i>stag</i> is present, the assembler puts the structure members in the global symbol table with the value of their absolute offset from the top of the structure. A <i>stag</i> is optional for <b>.struct</b>, but is required for <b>.tag</b>.</li> <li>• The <i>element</i> is one of the following descriptors: <b>.string</b>, <b>.byte</b>, <b>.char</b>, <b>.int</b>, <b>.half</b>, <b>.short</b>, <b>.word</b>, <b>.long</b>, <b>.double</b>, <b>.float</b>, <b>.tag</b>, or <b>.field</b>. All of these except <b>.tag</b> are typical directives that initialize memory. Following a <b>.struct</b> directive, these directives describe the structure element's size. They do not allocate memory. A <b>.tag</b> directive is a special case because <i>stag</i> must be used (as in the definition of <i>stag</i>).</li> <li>• The <i>expr</i> is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.</li> <li>• The <i>expr<sub>n/N</sub></i> is an optional expression for the number of elements described. This value defaults to 1. A <b>.string</b> element is considered to be one byte in size, and a <b>.field</b> element is one bit.</li> <li>• The <i>mem<sub>n/N</sub></i> is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.</li> <li>• The <i>size</i> is an optional label for the total size of the structure.</li> <li>• The <i>stag</i> is the structure's tag. Its value is associated with the beginning of the structure. If no <i>stag</i> is present, the assembler puts the structure members in the global symbol table with the value of their absolute offset from the top of the structure. A <i>stag</i> is optional for <b>.struct</b>, but is required for <b>.tag</b>.</li> </ul>		
	<p><b>Example</b> This example illustrates a structure in C that will be accessed in assembly code.</p>		

---

**.cstruct/.cunion/.endstruct/.endunion/.tag — Declare C Structure Type**


---

```

; typedef struct STRUCT1
; {      int i0;          /* offset 0 */
;      short s0;         /* offset 4 */
; } struct1;             /* size 8, alignment 4 */
;
; typedef struct STRUCT2
; {      struct1 st1; /* offset 0 */
;      short s1;     /* offset 8 */
; } struct2;         /* size 12, alignment 4 */
;
; The structure will get the following offsets once
; the C compiler lays out the structure elements according
; to the C standard rules:
;
; offsetof(struct1, i0) = 0
; offsetof(struct1, s0) = 4
; sizeof(struct1)      = 8
;
; offsetof(struct2, s1) = 0
; offsetof(struct2, il) = 8
; sizeof(struct2)      = 12
;
;
; Attempts to replicate this structure in assembly using the
; .struct/.union directive will not create the correct offsets
; because the assembler tries to use the most compact arrangement:

struct1      .struct
i0           .int                ; bytes 0-3
s0           .short              ; bytes 4-5
struct1len   .endstruct          ; size 6, alignment 4

struct2      .struct
st1          .tag struct1        ; bytes 0-5
s1           .short              ; bytes 6-7
endstruct2   .endstruct          ; size 8, alignment 4

                .sect "data1"
                .word struct1.i0      ; 0
                .word struct1.s0      ; 4
                .word struct1len      ; 6

                .sect "data2"
                .word struct2.st1     ; 0
                .word struct2.s1     ; 6
                .word endstruct2     ; 8
;
; The .cstruct/.cunion directives will calculate
; the offsets in the same manner as the C compiler. The
; resulting assembly structure can be used to access the
; elements of the C structure. Notice the different in
; the offsets from those structures defined via .struct
; above, and compare them to the offsets for the C code.

cstruct1 .cstruct
i0       .int                ; bytes 0-3
s0       .short              ; bytes 4-5
cstruct1len .endstruct      ; size 8, alignment 4

cstruct2 .cstruct
st1      .tag cstruct1       ; bytes 0-7
s1       .short              ; bytes 8-9
cendstruct2 .endstruct      ; size 12, alignment 4

                .sect "data3"
                .word cstruct1.i0, struct1.i0 ; 0

```

---



---

**.cstruct/.cunion/.endstruct/.endunion/.tag** — *Declare C Structure Type*

---

```
.word cstruct1.s0, struct1.s0 ; 4
.word cstruct1len, struct1len ; 8

.sect "data4"
.word cstruct2.st1, struct2.st1 ; 0
.word cstruct2.s1, struct2.s1 ; 8
.word cendstruct2, endstruct2 ; 12
```

**.data****Assemble Into the .data Section****Syntax****.data****Description**

The **.data** directive tells the assembler to begin assembling source code into the **.data** section; **.data** becomes the current section. The **.data** section is normally used to contain tables of data or preinitialized variables.

For more information about COFF sections, see [Chapter 2, Introduction to Common Object File Format](#).

**Example**

In this example, code is assembled into the **.data** and **.text** sections.

```

1          *****
2          **      Reserve space in .data      **
3          *****
4 00000000          .data
5 00000000          .space 0CCh
6
7          *****
8          **      Assemble into .text      **
9          *****
10 00000000          .text
11 00000000 00800358      ABS      A0,A1
12
13          *****
14          **      Assemble into .data      **
15          *****
16 000000cc          table: .data
17 000000cc FFFFFFFF          .word   -1
18 000000d0 000000FF          .byte   0FFh
19
20          *****
21          **      Assemble into .text      **
22          *****
23 00000004          .text
24 00000004 008001A0      MV      A0,A1
25
26          *****
27          **      Resume assembling into the .data section **
28          *****
29 000000d1          .data
30 000000d4 00000000      coeff   .word   00h,0ah,0bh
    000000d8 0000000A
    000000dc 0000000B

```

## **.double**

### **Initialize Double-Precision Floating-Point Value**

#### **Syntax**

**.double** *value*<sub>1</sub> [, ... , *value*<sub>*n*</sub>]

#### **Description**

The **.double** directive places the IEEE double-precision floating-point representation of one or more floating-point values into the current section. Each *value* must be a floating-point constant or a symbol that has been equated to a floating-point constant. Each constant is converted to a floating-point value in IEEE double-precision 64-bit format. Double-precision floating-point constants are aligned to a double word boundary.

The 64-bit value is stored in the format shown in [Figure 4-5](#).

**Figure 4-5. Double-Precision Floating-Point Format**



When you use **.double** in a **.struct/.endstruct** sequence, **.double** defines a member's size; it does not initialize memory. For more information, see the [.struct/.endstruct/.tag](#) topic.

#### **Example**

This example shows the **.double** directive.

```
1 00000000 2C280291          .double -2.0e25
   00000004 C5308B2A
2 00000008 00000000          .double 6
   0000000c 40180000
3 00000010 00000000          .double 456
   00000014 407C8000
```

**.drlist/.drnolist****Control Listing of Directives**

---

**Syntax****.drlist****.drnolist****Description**

Two directives enable you to control the printing of assembler directives to the listing file:

The **.drlist** directive enables the printing of all directives to the listing file.

The **.drnolist** directive suppresses the printing of the following directives to the listing file. The **.drnolist** directive has no affect within macros.

- **.asg**
- **.break**
- **.emsg**
- **.eval**
- **.fclist**
- **.fcnolist**
- **.mlist**
- **.mmsg**
- **.mnolist**
- **.sslist**
- **.ssnolist**
- **.var**
- **.wmsg**

By default, the assembler acts as if the **.drlist** directive had been specified.

**Example**

This example shows how **.drnolist** inhibits the listing of the specified directives.

**Source file:**

```
.length 65
.width 85
.asg 0, x
.loop 2
.eval x+1, x
.endloop

.drnolist
.length 55
.width 95
.asg 1, x
.loop 3
.eval x+1, x
.endloop
```

**Listing file:**

```

3          .asg 0, x
4          .loop 2
5          .eval x+1, x
6          .endloop
1          .eval 0+1, x
1          .eval 1+1, x

7
8          .drnolist
12         .loop 3
13         .eval x+1, x
14         .endloop
```



## Directives Reference

---

```
1                                     .endif
```

```
1 Error, No Warnings
```

In addition, the following messages are sent to standard output by the assembler:

```
*** ERROR!   line 12:  ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
               .msg    "ERROR -- MISSING PARAMETER"    ]]
```

```
1 Assembly Error, No Assembly Warnings
Errors in source - Assembler Aborted
```

## **.end**

### **End Assembly**

---

#### Syntax

**.end**

#### Description

The **.end** directive is optional and terminates assembly. The assembler ignores any source statements that follow a **.end** directive. If you use the **.end** directive, it must be the last source statement of a program.

This directive has the same effect as an end-of-file character. You can use **.end** when you are debugging and you want to stop assembling at a specific point in your code.

---

#### Ending a Macro

**Note:** Do not use the **.end** directive to terminate a macro; use the **.endm** macro directive instead.

---

#### Example

This example shows how the **.end** directive terminates assembly. If any source statements follow the **.end** directive, the assembler ignores them.

##### Source file:

```
start:  .text
        ZERO    A0
        ZERO    A1
        ZERO    A3
        .end
        ZERO    A4
```

##### Listing file:

```
1 00000000          start:  .text
2 00000000 000005E0      ZERO    A0
3 00000004 008425E0      ZERO    A1
4 00000008 018C65E0      ZERO    A3
5                      .end
```

**.fclist/.fcnolist**
**Control Listing of False Conditional Blocks**
**Syntax**
**.fclist**
**.fcnolist**
**Description**

Two directives enable you to control the listing of false conditional blocks:

The **.fclist** directive allows the listing of false conditional blocks (conditional blocks that do not produce code).

The **.fcnolist** directive suppresses the listing of false conditional blocks until a **.fclist** directive is encountered. With **.fcnolist**, only code in conditional blocks that are actually assembled appears in the listing. The **.if**, **.elseif**, **.else**, and **.endif** directives do not appear.

By default, all conditional blocks are listed; the assembler acts as if the **.fclist** directive had been used.

**Example**

This example shows the assembly language and listing files for code with and without the conditional blocks listed.

**Source file:**

```
a  .set      0
b  .set      1
   .fclist   ; list false conditional blocks
   .if      a
   MVK      5,A0
   .else
   MVK      0,A0
   .endif
   .fcnolist ; do not list false conditional blocks
   .if      a
   MVK      5,A0
   .else
   MVK      0,A0
   .endif
```

**Listing file:**

```
1          00000000 a  .set      0
2          00000001 b  .set      1
3                                     .fclist   ; list false conditional blocks
4                                     .if      a
5                                     MVK      5,A0
6                                     .else
7 00000000 00000028   MVK      0,A0
8                                     .endif
9                                     .fcnolist ; do not list false conditional blocks
13 00000004 00000028   MVK      0,A0
```

**.field****Initialize Field****Syntax****.field** *value* [, *size in bits*]**Description**

The **.field** directive initializes a multiple-bit field within a single word of memory. This directive has two operands:

- The *value* is a required parameter; it is an expression that is evaluated and placed in the field. The value must be absolute.
- The *size in bits* is an optional parameter; it specifies a number from 1 to 32, which is the number of bits in the field. If you do not specify a size, the assembler assumes the size is 32 bits. If you specify a value that cannot fit in *size in bits*, the assembler truncates the value and issues a warning message. For example, **.field 3,1** causes the assembler to truncate the value 3 to 1; the assembler also prints the message:

```
*** WARNING! line 21: W0001: Field value truncated to 1
      .field 3, 1
```

Successive **.field** directives pack values into the specified number of bits starting at the current 32-bit slot. Fields are packed starting at the least significant bit (bit 0), moving toward the most significant bit (bit 31) as more fields are added. If the assembler encounters a field size that does not fit in the current 32-bit word, it fills the remaining bits of the current byte with 0s, increments the SPC to the next word boundary, and begins packing fields into the next word.

You can use the **.align** directive to force the next **.field** directive to begin packing into a new word.

If you use a label, it points to the byte that contains the specified field.

When you use **.field** in a **.struct/.endstruct** sequence, **.field** defines a member's size; it does not initialize memory. For more information, see the [.struct/.endstruct/.tag topic](#).

**Example**

This example shows how fields are packed into a word. The SPC does not change until a word is filled and the next word is begun. [Figure 4-6](#) shows how the directives in this example affect memory.

```

1          *****
2          **      Initialize a 24-bit field.      **
3          *****
4 00000000 00BBCCDD          .field 0BBCCDDh, 24
5
6          *****
7          **      Initialize a 5-bit field        **
8          *****
9 00000000 0ABBCCDD          .field 0Ah, 5
10
11         *****
12         **      Initialize a 4-bit field        **
13         **      in a new word.                  **
14         *****
15 00000004 0000000C          .field 0Ch, 4
16
17         *****
18         **      Initialize a 3-bit field        **
19         *****
20 00000004 0000001C x:      .field 01h, 3
21
22         *****
23         **      Initialize a 32-bit field      **
24         **      relocatable field in the      **
25         **      next word                      **
26         *****
27 00000008 00000004'          .field x
```





**.global/.def/.ref****Identify Global Symbols****Syntax****.global** *symbol*<sub>1</sub>[, ..., *symbol*<sub>*n*</sub>]**.def** *symbol*<sub>1</sub>[, ..., *symbol*<sub>*n*</sub>]**.ref** *symbol*<sub>1</sub>[, ..., *symbol*<sub>*n*</sub>]**Description**

Three directives identify global symbols that are defined externally or can be referenced externally:

The **.def** directive identifies a symbol that is defined in the current module and can be accessed by other files. The assembler places this symbol in the symbol table.

The **.ref** directive identifies a symbol that is used in the current module but is defined in another module. The linker resolves this symbol's definition at link time.

The **.global** directive acts as a **.ref** or a **.def**, as needed.

A global *symbol* is defined in the same manner as any other symbol; that is, it appears as a label or is defined by the **.set**, **.equ**, **.bss**, or **.usect** directive. As with all symbols, if a global symbol is defined more than once, the linker issues a multiple-definition error. The **.ref** directive always creates a symbol table entry for a symbol, whether the module uses the symbol or not; **.global**, however, creates an entry only if the module actually uses the symbol.

A symbol can be declared global for either of two reasons:

- If the symbol is *not defined in the current module* (which includes macro, copy, and include files), the **.global** or **.ref** directive tells the assembler that the symbol is defined in an external module. This prevents the assembler from issuing an unresolved reference error. At link time, the linker looks for the symbol's definition in other modules.
- If the symbol is *defined in the current module*, the **.global** or **.def** directive declares that the symbol and its definition can be used externally by other modules. These types of references are resolved at link time.

**Example**

This example shows four files. The **file1.lst** and **file2.lst** refer to each other for all symbols used; **file3.lst** and **file4.lst** are similarly related.

The **file1.lst** and **file3.lst** files are equivalent. Both files define the symbol **INIT** and make it available to other modules; both files use the external symbols **X**, **Y**, and **Z**. Also, **file1.lst** uses the **.global** directive to identify these global symbols; **file3.lst** uses **.ref** and **.def** to identify the symbols.

The **file2.lst** and **file4.lst** files are equivalent. Both files define the symbols **X**, **Y**, and **Z** and make them available to other modules; both files use the external symbol **INIT**. Also, **file2.lst** uses the **.global** directive to identify these global symbols; **file4.lst** uses **.ref** and **.def** to identify the symbols.

**file1.lst**

```

1                      ; Global symbol defined in this file
2                      .global  INIT
3                      ; Global symbols defined in file2.lst
4                      .global  X, Y, Z
5 00000000             INIT:
6 00000000 00902058      ADD.L1 0x01,A4,A1
7 00000004 00000000!      .word  X
8                      ;
9                      ;
10                     ;
11                     .end
```

**file2.lst**

```

1          ; Global symbols defined in this file
2          .global X, Y, Z
3          ; Global symbol defined in file1.lst
4          .global INIT
5          00000001 X:      .set    1
6          00000002 Y:      .set    2
7          00000003 Z:      .set    3
8 00000000 00000000!      .word    INIT
9          ;
10         ;
11         ;
12         .end

```

**file3.lst**

```

1          ; Global symbol defined in this file
2          .def    INIT
3          ; Global symbols defined in file4.lst
4          .ref    X, Y, Z
5 00000000      INIT:
6 00000000 00902058      ADD.L1 0x01,A4,A1
7 00000004 00000000!      .word    X
8          ;
9          ;
10         ;
11         .end

```

**file4.lst**

```

1          ; Global symbols defined in this file
2          .def    X, Y, Z
3          ; Global symbol defined in file3.lst
4          .ref    INIT
5          00000001 X:      .set    1
6          00000002 Y:      .set    2
7          00000003 Z:      .set    3
8 00000000 00000000!      .word    INIT
9          ;
10         ;
11         ;
12         .end

```

## **.half/.short/.uhalf/.ushort**

### **Initialize 16-Bit Integers**

**Syntax**

```
.half value1[, ... , valuen]
.short value1[, ... , valuen]
.uhalf value1[, ... , valuen]
.ushort value1[, ... , valuen]
```

**Description**

The **.half**, **.uhalf**, **.short**, and **.ushort** directives place one or more values into consecutive halfwords in the current section. Each value is placed in a 2-byte slot by itself. A *value* can be either:

- An expression that the assembler evaluates and treats as a 16-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant eight bits of a 16-bit field, which is padded with 0s.

The assembler truncates values greater than 16 bits. You can use as many values as fit on a single line, but the total line length cannot exceed 200 characters.

If you use a label with **.half** or **.short**, it points to the location where the assembler places the first byte.

The **.half** and **.short** directives perform a halfword (16-bit) alignment before data is written to the section. This guarantees that data resides on a 16-bit boundary.

When you use **.half** or **.short** in a **.struct/.endstruct** sequence, they define a member's size; they do not initialize memory. For more information, see the [.struct/.endstruct/.tag topic](#).

**Example**

In this example, **.half** is used to place 16-bit values (10, -1, abc, and a) into consecutive halfwords in memory; **.short** is used to place 16-bit values (8, -3, def, and b) into consecutive halfwords in memory. The label STRN has the value 100ch, which is the location of the first initialized halfword for **.short**.

```
1 00000000 .space 100h * 16
2 00001000 0000000A .half 10, -1, "abc", 'a'
   00001002 0000FFFF
   00001004 00000061
   00001006 00000062
   00001008 00000063
   0000100a 00000061
3 0000100c 00000008 STRN .short 8, -3, "def", 'b'
   0000100e 0000FFFD
   00001010 00000064
   00001012 00000065
   00001014 00000066
   00001016 00000062
```

## **.if/.elseif/.else/.endif**

### **Assemble Conditional Blocks**

#### **Syntax**

```
.if well-defined expression
[.elseif well-defined expression]
[.else]
.endif
```

#### **Description**

Four directives provide conditional assembly:

The **.if** directive marks the beginning of a conditional block. The *well-defined expression* is a required parameter.

- If the expression evaluates to true (nonzero), the assembler assembles the code that follows the expression (up to a **.elseif**, **.else**, or **.endif**).
- If the expression evaluates to false (0), the assembler assembles code that follows a **.elseif** (if present), **.else** (if present), or **.endif** (if no **.elseif** or **.else** is present).

The **.elseif** directive identifies a block of code to be assembled when the **.if** expression is false (0) and the **.elseif** expression is true (nonzero). When the **.elseif** expression is false, the assembler continues to the next **.elseif** (if present), **.else** (if present), or **.endif** (if no **.elseif** or **.else** is present). The **.elseif** directive is optional in the conditional block, and more than one **.elseif** can be used. If an expression is false and there is no **.elseif** statement, the assembler continues with the code that follows a **.else** (if present) or a **.endif**.

The **.else** directive identifies a block of code that the assembler assembles when the **.if** expression and all **.elseif** expressions are false (0). The **.else** directive is optional in the conditional block; if an expression is false and there is no **.else** statement, the assembler continues with the code that follows the **.endif**.

The **.endif** directive terminates a conditional block.

The **.elseif** and **.else** directives can be used in the same conditional assembly block, and the **.elseif** directive can be used more than once within a conditional assembly block.

For information about relational operators, see [Section 3.9.4, Conditional Expressions](#).

#### **Example**

This example shows conditional assembly:

```

1          00000001  SYM1  .set    1
2          00000002  SYM2  .set    2
3          00000003  SYM3  .set    3
4          00000004  SYM4  .set    4
5
6          If_4:      .if  SYM4 = SYM2 * SYM2
7 00000000 00000004  .byte  SYM4          ; Equal values
8                                     .else
9                                     .byte  SYM2 * SYM2      ; Unequal values
10                                    .endif
11
12          If_5:      .if  SYM1 <= 10
13 00000001 0000000A  .byte  10           ; Less than / equal
14                                     .else
15                                     .byte  SYM1           ; Greater than
16                                    .endif
17
18          If_6:      .if  SYM3 * SYM2 != SYM4 + SYM2
19                                     .byte  SYM3 * SYM2      ; Unequal value
20                                     .else
21 00000002 00000008  .byte  SYM4 + SYM4      ; Equal values
22                                    .endif
23
24          If_7:      .if  SYM1 = SYM2
25                                     .byte  SYM1
```

## Directives Reference

```

26                                     .elseif SYM2 + SYM3 = 5
27 00000003 00000005               .byte  SYM2 + SYM3
28                                     .endif

```

## **.int/.long/.word/.uint/.uword**

### **Initialize 32-Bit Integers**

**Syntax**

```

.int value1[, ... , valuen]
.long value1[, ... , valuen]
.word value1[, ... , valuen]
.uint value1[, ... , valuen]
.uword value1[, ... , valuen]

```

**Description**

The **.int**, **.uint**, **.long**, **.word** and **.uword** directives place one or more values into consecutive words in the current section. Each value is placed in a 32-bit word by itself and is aligned on a word boundary. A *value* can be either:

- An expression that the assembler evaluates and treats as a 32-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant eight bits of a 32-bit field, which is padded with 0s.

A value can be either an absolute or a relocatable expression. If an expression is relocatable, the assembler generates a relocation entry that refers to the appropriate symbol; the linker can then correctly patch (relocate) the reference. This allows you to initialize memory with pointers to variables or labels.

You can use as many values as fit on a single line (200 characters). If you use a label with **.int**, **.long**, or **.word**, it points to the first word that is initialized.

When you use **.int**, **.long**, or **.word** directives in a **.struct/.endstruct** sequence, they define a member's size; they do not initialize memory. For more information, see the [.struct/.endstruct/.tag](#) topic.

**Example 1**

This example uses the **.int** directive to initialize words. Notice that the symbol SYMPTR puts the symbol's address in the object code and generates a relocatable reference (indicated by the - character appended to the object word).

```

1 00000000               .space 73h
2 00000000               .bss  PAGE, 128
3 00000080               .bss  SYMPTR, 3
4 00000074 003C12E4 INST: LDW.D2 *++B15[0],A0
5 00000078 0000000A       .int  10, SYMPTR, -1, 35 + 'a', INST
   0000007c 00000080-
   00000080 FFFFFFFF
   00000084 00000084
   00000088 00000074'

```

**Example 2**

This example initializes two 32-bit fields and defines DAT1 to point to the first location. The contents of the resulting 32-bit fields are FFFABCDh and 141h.

```

1 00000000 FFFABCD DAT1:  .long  0FFFFABCDh, 'A'+100h
   00000004 00000141

```

**Example 3**

This example initializes five words. The symbol WordX points to the first word.

```

1 00000000 00000C80 ;WordX  .word  3200,1+'AB',-'AF',0F410h,'A'
   00000004 00004242
   00000008 FFFFB9BF
   0000000c 0000F410
   00000010 00000041

```

### Data Size of longs

**Note:** For the C6000 C/C++ compiler, a long data value is 40 bits. For the C6000 assembler, a long data value is 32 bits. Therefore, the .long directive treats values assigned to it as 32-bit values.

## .label

### Create a Load-Time Address Label

#### Syntax

**.label** *symbol*

#### Description

The **.label** directive defines a special *symbol* that refers to the load-time address rather than the run-time address within the current section. Most sections created by the assembler have relocatable addresses. The assembler assembles each section as if it started at 0, and the linker relocates it to the address at which it loads and runs.

For some applications, it is desirable to have a section load at one address and run at a *different* address. For example, you may want to load a block of performance-critical code into slower memory to save space and then move the code to high-speed memory to run it. Such a section is assigned two addresses at link time: a load address and a run address. All labels defined in the section are relocated to refer to the run-time address so that references to the section (such as branches) are correct when the code runs.

The **.label** directive creates a special label that refers to the *load-time* address. This function is useful primarily to designate where the section was loaded for purposes of the code that relocates the section.

#### Example

This example shows the use of a load-time address label.

```
sect ".examp"
    .label examp_load ; load address of section
start:                ; run address of section
    <code>
finish:               ; run address of section end
    .label examp_end ; load address of section end
```

For more information about assigning run-time and load-time addresses in the linker, see [Section 7.9, Specifying a Section's Run-Time Address](#)

**.length/.width****Set Listing Page Size****Syntax****.length** [*page length*]**.width** [*page width*]**Description**

Two directives allow you to control the size of the output listing file.

The **.length** directive sets the page length of the output listing file. It affects the current and following pages. You can reset the page length with another **.length** directive.

- Default length: 60 lines. If you do not use the **.length** directive or if you use the **.length** directive without specifying the *page length*, the output listing length defaults to 60 lines.
- Minimum length: 1 line
- Maximum length: 32 767 lines

The **.width** directive sets the page width of the output listing file. It affects the next line assembled and the lines following. You can reset the page width with another **.width** directive.

- Default width: 132 characters. If you do not use the **.width** directive or if you use the **.width** directive without specifying a *page width*, the output listing width defaults to 132 characters.
- Minimum width: 80 characters
- Maximum width: 200 characters

The width refers to a full line in a listing file; the line counter value, SPC value, and object code are counted as part of the width of a line. Comments and other portions of a source statement that extend beyond the page width are truncated in the listing.

The assembler does not list the **.width** and **.length** directives.

**Example**

The following example shows how to change the page length and width.

```
*****
**          Page length = 65 lines          **
**          Page width = 85 characters       **
*****
          .length      65
          .width       85

*****
**          Page length = 55 lines          **
**          Page width = 100 characters     **
*****
          .length      55
          .width       100
```



## **.list/.nolist**

### **Start/Stop Source Listing**

#### **Syntax**

**.list**

**.nolist**

#### **Description**

Two directives enable you to control the printing of the source listing:

The **.list** directive allows the printing of the source listing.

The **.nolist** directive suppresses the source listing output until a **.list** directive is encountered. The **.nolist** directive can be used to reduce assembly time and the source listing size. It can be used in macro definitions to suppress the listing of the macro expansion.

The assembler does not print the **.list** or **.nolist** directives or the source statements that appear after a **.nolist** directive. However, it continues to increment the line counter. You can nest the **.list/.nolist** directives; each **.nolist** needs a matching **.list** to restore the listing.

By default, the source listing is printed to the listing file; the assembler acts as if the **.list** directive had been used. However, if you do not request a listing file when you invoke the assembler by including the **-al** option on the command line (see [Section 3.3, Invoking the Assembler](#)), the assembler ignores the **.list** directive.

#### **Example**

This example shows how the **.list** and **.nolist** directives turn the output listing on and off. The **.nolist**, the table: **.data** through **.byte** lines, and the **.list** directives do not appear in the listing file. Also, the line counter is incremented even when source statements are not listed.

##### **Source file:**

```
.data
.space 0CCh
.text
ABS      A0,A1

.nolist

table:   .data
        .word  -1
        .byte  0FFh

.list

        .text
MV       A0,A1
        .data
coeff    .word  00h,0ah,0bh
```

##### **Listing file:**

```
1 00000000          .data
2 00000000          .space 0CCh
3 00000000          .text
4 00000000 00800358  ABS      A0,A1
5
13
14 00000004          .text
15 00000004 008001A0  MV       A0,A1
16 000000d1          .data
17 000000d4 00000000  coeff    .word  00h,0ah,0bh
    000000d8 0000000A
    000000dc 0000000B
```

**.loop/.endloop/.break****Assemble Code Block Repeatedly**

---

**Syntax**

```
.loop [well-defined expression]
.break [well-defined expression]
.endloop
```

**Description**

Three directives allow you to repeatedly assemble a block of code:

The **.loop** directive begins a repeatable block of code. The optional expression evaluates to the loop count (the number of loops to be performed). If there is no *well-defined expression*, the loop count defaults to 1024, unless the assembler first encounters a **.break** directive with an expression that is true (nonzero) or omitted.

The **.break** directive, along with its expression, is optional. This means that when you use the **.loop** construct, you do not have to use the **.break** construct. The **.break** directive terminates a repeatable block of code only if the *well-defined expression* is true (nonzero) or omitted, and the assembler breaks the loop and assembles the code after the **.endloop** directive. If the expression is false (evaluates to 0), the loop continues.

The **.endloop** directive terminates a repeatable block of code; it executes when the **.break** directive is true (nonzero) or when the number of loops performed equals the loop count given by **.loop**.

**Example**

This example illustrates how these directives can be used with the **.eval** directive. The code in the first six lines expands to the code immediately following those six lines.

```

1          .eval      0,x
2          COEF  .loop
3          .word      x*100
4          .eval      x+1, x
5          .break     x = 6
6          .endloop
1          00000000 00000000 .word      0*100
1          .eval      0+1, x
1          .break     1 = 6
1          00000004 00000064 .word      1*100
1          .eval      1+1, x
1          .break     2 = 6
1          00000008 000000C8 .word      2*100
1          .eval      2+1, x
1          .break     3 = 6
1          0000000c 0000012C .word      3*100
1          .eval      3+1, x
1          .break     4 = 6
1          00000010 00000190 .word      4*100
1          .eval      4+1, x
1          .break     5 = 6
1          00000014 000001F4 .word      5*100
1          .eval      5+1, x
1          .break     6 = 6
```

## **.macro/.endm**

### **Define Macro**

#### **Syntax**

```
macname .macro [parameter1[, ... parametern]]
        model statements or macro directives
    .endm
```

#### **Description**

The **.macro** and **.endm** directives are used to define macros.

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file, in an `.include/.copy` file, or in a macro library.

<i>macname</i>	names the macro. You must place the name in the source statement's label field.
<b>.macro</b>	identifies the source statement as the first line of a macro definition. You must place <b>.macro</b> in the opcode field.
[ <i>parameters</i> ]	are optional substitution symbols that appear as operands for the <b>.macro</b> directive.
<i>model statements</i>	are instructions or assembler directives that are executed each time the macro is called.
<i>macro directives</i>	are used to control macro expansion.
<b>.endm</b>	marks the end of the macro definition.

Macros are explained in further detail in [Chapter 5, Macro Language](#).

## **.mlib**

### **Define Macro Library**

#### **Syntax**

```
.mlib ["filename"]
```

#### **Description**

The **.mlib** directive provides the assembler with the *filename* of a macro library. A macro library is a collection of files that contain macro definitions. The macro definition files are bound into a single file (called a library or archive) by the archiver.

Each file in a macro library contains one macro definition that corresponds to the name of the file. The *filename* of a macro library member must be the same as the macro name, and its extension must be `.asm`. The filename must follow host operating system conventions; it can be enclosed in double quotes. You can specify a full pathname (for example, `c:\320tools\macs.lib`). If you do not specify a full pathname, the assembler searches for the file in the following locations in the order given:

1. The directory that contains the current source file
2. Any directories named with the `-I` assembler option
3. Any directories specified by the `C6X_A_DIR` or `A_DIR` environment variable

For more information about the `-I` option, `C6X_A_DIR`, and `A_DIR`, see [Section 3.4, Naming Alternate Directories for Assembler Input](#).

When the assembler encounters a **.mlib** directive, it opens the library specified by the *filename* and creates a table of the library's contents. The assembler enters the names of the individual library members into the opcode table as library entries. This redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table. The assembler expands the library entry in the same way it expands other macros, but it does not place the source code into the listing. Only macros that are actually called from the library are extracted, and they are extracted only once.

For more information on macros and macro libraries, see [Chapter 5, Macro Language](#).

#### **Example**

This example creates a macro library that defines two macros, `inc1` and `dec1`. The file

## Directives Reference

inc1.asm contains the definition of inc1, and dec1.asm contains the definition of dec1.

inc1.asm	dec1.asm
<pre>* Macro for incrementing inc1 .macro A     ADD A,1,A .endm</pre>	<pre>* Macro for decrementing dec1 .macro A     SUB A,1,A .endm</pre>

Use the archiver to create a macro library:

```
ar6x -a mac inc1.asm dec1.asm
```

Now you can use the .mlib directive to reference the macro library and define the inc1 and dec1 macros:

```

1                                     .mlib    "mac.lib"
2
3                                     * Macro Call
4 00000000                          inc1      A0
1 00000000 000021A0                  ADD      A0,1,A0
5
6                                     * Macro Call
7 00000004                          dec1      B0
1 00000004 0003E1A2                  SUB      B0,1,B0
```

**.mlist/.mnolist****Start/Stop Macro Expansion Listing****Syntax****.mlist****.mnolist****Description**

Two directives enable you to control the listing of macro and repeatable block expansions in the listing file:

The **.mlist** directive allows macro and .loop/.endloop block expansions in the listing file.

The **.mnolist** directive suppresses macro and .loop/.endloop block expansions in the listing file.

By default, the assembler behaves as if the .mlist directive had been specified.

For more information on macros and macro libraries, see [Chapter 5, Macro Language](#). For more information, see the [.loop/.break/.endloop](#) topic .

**Example**

This example defines a macro named STR\_3. The first time the macro is called, the macro expansion is listed (by default). The second time the macro is called, the macro expansion is not listed, because a .mnolist directive was assembled. The third time the macro is called, the macro expansion is again listed because a .mlist directive was assembled.

```

1                                     STR_3 .macro    P1, P2, P3
2                                     .string ":p1:", ":p2:", ":p3:"
3                                     .endm
4
5 00000000                          STR_3 "as", "I", "am"
1 00000000 0000003A                  .string ":p1:", ":p2:", ":p3:"
00000001 00000070
00000002 00000031
00000003 0000003A
00000004 0000003A
00000005 00000070
00000006 00000032
00000007 0000003A
00000008 0000003A
00000009 00000070
0000000a 00000033
0000000b 0000003A
6                                     .mnolist
```

```

7 0000000c          STR_3 "as", "I", "am"
8                  .mlist
9 00000018          STR_3 "as", "I", "am"
1 00000018 0000003A  .string ":p1:", ":p2:", ":p3:"
   00000019 00000070
   0000001a 00000031
   0000001b 0000003A
   0000001c 0000003A
   0000001d 00000070
   0000001e 00000032
   0000001f 0000003A
   00000020 0000003A
   00000021 00000070
   00000022 00000033
   00000023 0000003A

```

## **.newblock**

### **Terminate Local Symbol Block**

#### **Syntax**

**.newblock**

#### **Description**

The **.newblock** directive undefines any local labels currently defined. Local labels, by nature, are temporary; the **.newblock** directive resets them and terminates their scope.

A local label is a label in the form  $\$n$ , where  $n$  is a single decimal digit, or *name?*, where *name* is a legal symbol name. Unlike other labels, local labels are intended to be used locally, cannot be used in expressions, and do not qualify for branch expansion if used with a branch. They can be used only as operands in 8-bit jump instructions. Local labels are not included in the symbol table.

After a local label has been defined and (perhaps) used, you should use the **.newblock** directive to reset it. The **.text**, **.data**, and **.sect** directives also reset local labels. Local labels that are defined within an include file are not valid outside of the include file.

For more information on the use of local labels, see [Section 3.8.2, Local Labels](#).

#### **Example**

This example shows how the local label  $\$1$  is declared, reset, and then declared again.

```

1                  .global table1, table2
2
3 00000000 00000028!   MVKL    table1,A0
4 00000004 00000068!   MVKH    table1,A0
5 00000008 008031A9    MVK     99, A1
6 0000000c 010848C0 ||  ZERO    A2
7
8 00000010 80000212 $1:[A1] B    $1
9 00000014 01003674    STW     A2, *A0++
10 00000018 0087E1A0    SUB     A1,1,A1
11 0000001c 00004000    NOP     3
12
13                  .newblock ; undefine $1
14
15 00000020 00000028!   MVKL    table2,A0
16 00000024 00000068!   MVKH    table2,A0
17 00000028 008031A9    MVK     99, A1
18 0000002c 010829C0 ||  SUB     A2,1,A2
19
20 00000030 80000212 $1:[A1] B    $1
21 00000034 01003674    STW     A2, *A0++
22 00000038 0087E1A0    SUB     A1,1,A1
23 0000003c 00004000    NOP     3

```

**.nocmp*****Do Not Utilize 16-Bit Instructions in Section***

---

**Syntax****.nocmp****Description**

The C6400+ **.nocmp** directive instructs the compiler to not utilize 16-bit instructions for the code section **.nocmp** appears in. The **.nocmp** directive can appear anywhere in the section.

**Example**

In the example, the section one is not compressed, whereas section two is compressed.

```
.sect "one"
LDW *A4, A5
LDW *B4, A5
.nocmp
NOP 4
ADD A4, A5, A6
ADD B4, B5, B6
NOP
...

.sect "two"
ADD A4, A5, A6
NOP
NOP
...
```

**.option*****Select Listing Options***

---

**Syntax****.option** *option*<sub>1</sub>[, *option*<sub>2</sub>, . . .]**Description**

The **.option** directive selects options for the assembler output listing. The *options* must be separated by commas; each option selects a listing feature. These are valid options:

- A** turns on listing of all directives and data, and subsequent expansions, macros, and blocks.
- B** limits the listing of **.byte** and **.char** directives to one line.
- D** turns off the listing of certain directives (same effect as **.drnolist**).
- H** limits the listing of **.half** and **.short** directives to one line.
- L** limits the listing of **.long** directives to one line.
- M** turns off macro expansions in the listing.
- N** turns off listing (performs **.nolist**).
- O** turns on listing (performs **.list**).
- R** resets the **B**, **H**, **L**, **M**, **T**, and **W** directives (turns off the limits of **B**, **H**, **L**, **M**, **T**, and **W**).
- T** limits the listing of **.string** directives to one line.
- W** limits the listing of **.word** and **.int** directives to one line.
- X** produces a cross-reference listing of symbols. You can also obtain a cross-reference listing by invoking the assembler with the **-x** option (see [Section 3.3](#)).

Options *are not* case sensitive.

**Example**

This example shows how to limit the listings of the **.byte**, **.char**, **.int**, **.word**, and **.string** directives to one line each.

```
1 *****
2 ** Limit the listing of .byte, .char, **
3 ** .int, .word, and .string **
```

```

4          **      directives to 1 line each.      **
5          ****
6          .option B, W, T
7 00000000 000000BD      .byte  -'C', 0B0h, 5
8 00000003 000000BC      .char   -'D', 0C0h, 6
9 00000008 0000000A      .int     10, 35 + 'a', "abc"
10 0000001c AABBCDD      .long    0AABCCDDh, 536 + 'A'
    00000020 00000259
11 00000024 000015AA      .word    5546, 78h
12 0000002c 00000052      .string  "Registers"
13
14          ****
15          **      Reset the listing options.      **
16          ****
17          .option R
18 00000035 000000BD      .byte  -'C', 0B0h, 5
    00000036 000000B0
    00000037 00000005
19 00000038 000000BC      .char   -'D', 0C0h, 6
    00000039 000000C0
    0000003a 00000006
20 0000003c 0000000A      .int     10, 35 + 'a', "abc"
    00000040 00000084
    00000044 00000061
    00000048 00000062
    0000004c 00000063
21 00000050 AABBCDD      .long    0AABCCDDh, 536 + 'A'
    00000054 00000259
22 00000058 000015AA      .word    5546, 78h
    0000005c 00000078
23 00000060 00000052      .string  "Registers"
    00000061 00000065
    00000062 00000067
    00000063 00000069
    00000064 00000073
    00000065 00000074
    00000066 00000065
    00000067 00000072
    00000068 00000073

```

## **.page**

### ***Eject Page in Listing***

#### **Syntax**

**.page**

#### **Description**

The **.page** directive produces a page eject in the listing file. The **.page** directive is not printed in the source listing, but the assembler increments the line counter when it encounters the **.page** directive. Using the **.page** directive to divide the source listing into logical divisions improves program readability.

#### **Example**

This example shows how the **.page** directive causes the assembler to begin a new page of the source listing.

##### **Source file:**

```

        .title    "**** Page Directive Example ****"
;
;
;
        .page

```

##### **Listing file:**

```

TMS320C6x COFF Assembler      Version x.xx      Tue Apr 14 17:16:51 1997
Copyright ©) 1996-1997 Texas Instruments Incorporated
**** Page Directive Example ****

```

PAGE 1

```

2          ;      .
3          ;      .
4          ;      .
TMS320C6x COFF Assembler Version x.xx      Tue Apr 14 17:16:51 1997
Copyright ©) 1996-1997 Texas Instruments Incorporated
**** Page Directive Example ****
PAGE      2

```

No Errors, No Warnings

## **.sect** *Assemble Into Named Section*

---

**Syntax** `.sect "section name"`

**Description** The **.sect** directive defines a named section that can be used like the default `.text` and `.data` sections. The **.sect** directive tells the assembler to begin assembling source code into the named section.

The *section name* identifies the section. The section name is significant to 200 characters and must be enclosed in double quotes. A section name can contain a subsection name in the form *section name:subsection name*.

For more information about COFF sections, see [Chapter 2, Introduction to Common Object File Format](#).

**Example** This example defines one special-purpose section, `vars`, and assembles code into it.

```

1          *****
2          **   Begin assembling into .text section.   **
3          *****
4 00000000          .text
5 00000000 000005E0          ZERO      A0
6 00000004 008425E0          ZERO      A1
7
8          *****
9          **   Begin assembling into vars section.   **
10         *****
11 00000000          .sect "vars"
12 00000000 4048F5C3 pi      .float    3.14
13 00000004 000007D0 max     .int      2000
14 00000008 00000001 min     .int      1
15
16         *****
17         **   Resume assembling into .text section.   **
18         *****
19 00000008          .text
20 00000008 010000A8          MVK       1,A2
21 0000000c 018000A8          MVK       1,A3
22
23         *****
24         **   Resume assembling into vars section.   **
25         *****
26 0000000c          .sect "vars"
27 0000000c 00000019 count    .short   25

```



## **.set/.equ**

## **Define Assembly-Time Constant**

### **Syntax**

*symbol* **.set** *value*

*symbol* **.equ** *value*

### **Description**

The **.set** and **.equ** directives equate a constant value to a symbol. The symbol can then be used in place of a value in assembly source. This allows you to equate meaningful names with constants and other values. The **.set** and **.equ** directives are identical and can be used interchangeably.

- The *symbol* is a label that must appear in the label field.
- The *value* must be a well-defined expression, that is, all symbols in the expression must be previously defined in the current source module.

Undefined external symbols and symbols that are defined later in the module cannot be used in the expression. If the expression is relocatable, the symbol to which it is assigned is also relocatable.

The value of the expression appears in the object field of the listing. This value is not part of the actual object code and is not written to the output file.

Symbols defined with **.set** or **.equ** can be made externally visible with the **.def** or **.global** directive (see the [.global/.def/.ref topic](#)). In this way, you can define global absolute constants.

### **Example**

This example shows how symbols can be assigned with **.set** and **.equ**.

```

1          *****
2          **   Equate symbol AUX_R1 to register A1   **
3          **   and use it instead of the register.   **
4          *****
5          00000001  AUX_R1 .set A1
6 00000000 00B802D4      STH      AUX_R1, *+B14
7
8          *****
9          **   Set symbol index to an integer expr.   **
10         **   and use it as an immediate operand.   **
11         *****
12         00000035  INDEX .equ 100/2 +3
13 00000004 01001AD0      ADDK      INDEX, A2
14
15         *****
16         **   Set symbol SYMTAB to a relocatable expr. **
17         **   and use it as a relocatable operand.   **
18         *****
19 00000008 0000000A  LABEL .word 10
20         00000009' SYMTAB .set LABEL + 1
21
22         *****
23         **   Set symbol NSYMS equal to the symbol   **
24         **   INDEX and use it as you would INDEX.   **
25         *****
26         00000035  NSYMS .set INDEX
27 0000000c 00000035      .word  NSYMS

```

**.space/.bes****Reserve Space****Syntax**

[label] **.space** *size in bytes*

[label] **.bes** *size in bytes*

**Description**

The **.space** and **.bes** directives reserve the number of bytes given by *size in bytes* in the current section and fill them with 0s. The section program counter is incremented to point to the word following the reserved space.

When you use a label with the **.space** directive, it points to the *first* byte reserved. When you use a label with the **.bes** directive, it points to the *last* byte reserved.

**Example**

This example shows how memory is reserved with the **.space** and **.bes** directives.

```

1          *****
2          **      Begin assembling into the .text section.      **
3          *****
4 00000000          .text
5          *****
6          **      Reserve 0F0 bytes (60 words in .text section).  **
7          *****
8 00000000          .space 0F0h
9 000000f0 00000100          .word 100h, 200h
   000000f4 00000200
10         *****
11         **      Begin assembling into the .data section.      **
12         *****
13 00000000          .data
14 00000000 00000049          .string "In .data"
   00000001 0000006E
   00000002 00000020
   00000003 0000002E
   00000004 00000064
   00000005 00000061
   00000006 00000074
   00000007 00000061
15         *****
16         **      Reserve 100 bytes in the .data section;      **
17         **      RES_1 points to the first word                **
18         **      that contains reserved bytes.                  **
19         *****
20 00000008          RES_1: .space 100
21 0000006c 0000000F          .word 15
22 00000070 00000008"          .word RES_1
23         *****
24         **      Reserve 20 bytes in the .data section;      **
25         **      RES_2 points to the last word                **
26         **      that contains reserved bytes.                  **
27         *****
28 00000087          RES_2: .bes 20
29 00000088 00000036          .word 36h
30 0000008c 00000087"          .word RES_2

```

---

**.sslist/.ssnolist      Control Listing of Substitution Symbols**


---

**Syntax**
**.sslist**
**.ssnolist**
**Description**

Two directives allow you to control substitution symbol expansion in the listing file:

The **.sslist** directive allows substitution symbol expansion in the listing file. The expanded line appears below the actual source line.

The **.ssnolist** directive suppresses substitution symbol expansion in the listing file.

By default, all substitution symbol expansion in the listing file is suppressed; the assembler acts as if the **.ssnolist** directive had been used.

Lines with the pound (#) character denote expanded substitution symbols.

**Example**

This example shows code that, by default, suppresses the listing of substitution symbol expansion, and it shows the **.sslist** directive assembled, instructing the assembler to list substitution symbol code expansion.

```

1 00000000          .bss    x,4
2 00000004          .bss    y,4
3 00000008          .bss    z,4
4
5                  addm     .macro  src1,src2,dst
6                      LDW    *+B14(:src1:), A0
7                      LDW    *+B14(:src2:), A1
8                      NOP     4
9                      ADD     A0,A1,A0
10                     STW     A0,*+B14(:dst:)
11                     .endm
12
13 00000000          addm     x,y,z
1 00000000 0000006C- LDW     *+B14(x), A0
1 00000004 0080016C- LDW     *+B14(y), A1
1 00000008 00006000  NOP     4
1 0000000c 000401E0  ADD     A0,A1,A0
1 00000010 0000027C- STW     A0,*+B14(z)
14
15                  .sslist
16 00000014          addm     x,y,z
1 00000014 0000006C- LDW     *+B14(:src1:), A0
#                      LDW     *+B14(x), A0
1 00000018 0080016C- LDW     *+B14(:src2:), A1
#                      LDW     *+B14(y), A1
1 0000001c 00006000  NOP     4
1 00000020 000401E0  ADD     A0,A1,A0
1 00000024 0000027C- STW     A0,*+B14(:dst:)
#                      STW     A0,*+B14(z)
17

```

**.string****Initialize Text****Syntax**

**.string** {*expr*<sub>1</sub> | "*string*<sub>1</sub>"}, ... , {*expr*<sub>*n*</sub> | "*string*<sub>*n*</sub>"}

**Description**

The **.string** directive places 8-bit characters from a character string into the current section. The *expr* or *string* can be one of the following:

- An expression that the assembler evaluates and treats as an 8-bit signed number.
- A character string enclosed in double quotes. Each character in a string represents a separate value, and values are stored in consecutive bytes. The entire string *must* be enclosed in quotes.

The assembler truncates any values that are greater than eight bits. You can have up to 100 operands, but they must fit on a single source statement line.

If you use a label with **.string**, it points to the location of the first byte that is initialized.

When you use **.string** in a **.struct/.endstruct** sequence, **.string** defines a member's size; it does not initialize memory. For more information, see the [.struct/.endstruct/.tag](#) topic.

**Example**

In this example, 8-bit values are placed into consecutive bytes in the current section. The label **Str\_Ptr** has the value 0h, which is the location of the first initialized byte.

```

1 00000000 00000041 Str_Ptr:  .string  "ABCD"
   00000001 00000042
   00000002 00000043
   00000003 00000044
2 00000004 00000041          .string  41h, 42h, 43h, 44h
   00000005 00000042
   00000006 00000043
   00000007 00000044
3 00000008 00000041          .string  "Austin", "Houston"
   00000009 00000075
   0000000a 00000073
   0000000b 00000074
   0000000c 00000069
   0000000d 0000006E
   0000000e 00000048
   0000000f 0000006F
   00000010 00000075
   00000011 00000073
   00000012 00000074
   00000013 0000006F
   00000014 0000006E
4 00000015 00000030          .string  36 + 12

```

## **.struct/.endstruct/.tag**

### **Declare Structure Type**

Syntax	[stag]	<b>.struct</b>	[expr]
	[mem <sub>0</sub> ]	element	[expr <sub>0</sub> ]
	[mem <sub>1</sub> ]	element	[expr <sub>1</sub> ]
	.	.	.
	.	.	.
	.	.	.
	[mem <sub>n</sub> ]	<b>.tag</b> stag	[expr <sub>n</sub> ]
	.	.	.
	.	.	.
	.	.	.
	[mem <sub>N</sub> ]	element	[expr <sub>N</sub> ]
	[size]	<b>.endstruct</b>	
	label	<b>.tag</b>	stag
Description	<p>The <b>.struct</b> directive assigns symbolic offsets to the elements of a data structure definition. This allows you to group similar data elements together and let the assembler calculate the element offset. This is similar to a C structure or a Pascal record. The <b>.struct</b> directive does not allocate memory; it merely creates a symbolic template that can be used repeatedly.</p> <p>The <b>.endstruct</b> directive terminates the structure definition.</p> <p>The <b>.tag</b> directive gives structure characteristics to a <i>label</i>, simplifying the symbolic representation and providing the ability to define structures that contain other structures. The <b>.tag</b> directive does not allocate memory. The structure tag (<i>stag</i>) of a <b>.tag</b> directive must have been previously defined.</p> <p>Following are descriptions of the parameters used with the <b>.struct</b>, <b>.endstruct</b>, and <b>.tag</b> directives:</p> <ul style="list-style-type: none"> <li>• The <i>stag</i> is the structure's tag. Its value is associated with the beginning of the structure. If no <i>stag</i> is present, the assembler puts the structure members in the global symbol table with the value of their absolute offset from the top of the structure. A <i>.stag</i> is optional for <b>.struct</b>, but is required for <b>.tag</b>.</li> <li>• The <i>expr</i> is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.</li> <li>• The <i>mem<sub>n/N</sub></i> is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.</li> <li>• The <i>element</i> is one of the following descriptors: <b>.string</b>, <b>.byte</b>, <b>.char</b>, <b>.int</b>, <b>.half</b>, <b>.short</b>, <b>.word</b>, <b>.long</b>, <b>.double</b>, <b>.float</b>, <b>.tag</b>, or <b>.field</b>. All of these except <b>.tag</b> are typical directives that initialize memory. Following a <b>.struct</b> directive, these directives describe the structure element's size. They do not allocate memory. A <b>.tag</b> directive is a special case because <i>stag</i> must be used (as in the definition of <i>stag</i>).</li> <li>• The <i>expr<sub>n/N</sub></i> is an optional expression for the number of elements described. This value defaults to 1. A <b>.string</b> element is considered to be one byte in size, and a <b>.field</b> element is one bit.</li> <li>• The <i>size</i> is an optional label for the total size of the structure.</li> </ul>		

**.struct/.endstruct/.tag — Declare Structure Type****Directives That Can Appear in a .struct/.endstruct Sequence**

**Note:** The only directives that can appear in a .struct/.endstruct sequence are element descriptors, conditional assembly directives, and the .align directive, which aligns the member offsets on word boundaries. Empty structures are illegal.

The following examples show various uses of the .struct, .tag, and .endstruct directives.

**Example 1**

```

1          real_rec .struct                ; stag
2          00000000 nom .int                ; member1 = 0
3          00000004 den .int                ; member2 = 1
4          00000008 real_len .endstruct    ; real_len = 2
5
6 00000000 0080016C-          LDW  *+B14(real+real_rec.den), A1
7                                ; access structure
8
9 00000000                      .bss real, real_len    ; allocate mem rec
10

```

**Example 2**

```

11         cplx_rec .struct                ; stag
12         00000000 reali .tag real_rec    ; member1 = 0
13         00000008 imagi .tag real_rec    ; member2 = 2
14         00000010 cplx_len .endstruct    ; cplx_len = 4
15
16         complex .tag cplx_rec           ; assign structure
17                                ; attribute
18 00000008                      .bss complex, cplx_len ; allocate mem rec
19
20 00000004 0100046C-          LDW  *+B14(complex.imagi.nom), A2
21                                ; access structure
22 00000008 0100036C-          LDW  *+B14(complex.reali.den), A2
23                                ; access structure
24 0000000c 018C4A78          CMPEQ A2, A3, A3

```

**Example 3**

```

1          .struct                        ; no stag puts
2                                ; mems into global
3                                ; symbol table
4
5          00000000 X .byte                ; create 3 dim
6          00000001 Y .byte                ; templates
7          00000002 Z .byte
8          00000003 .endstruct

```

**Example 4**

```

1          bit_rec .struct                ; stag
2          00000000 stream .string 64
3          00000040 bit7 .field 7          ; bit7 = 64
4          00000040 bit1 .field 9          ; bit9 = 64
5          00000042 bit5 .field 10         ; bit5 = 64
6          00000044 x_int .byte            ; x_int = 68
7          00000045 bit_len .endstruct    ; length = 72
8
9          bits .tag bit_rec
10 00000000                      .bss bits, bit_len
11
12 00000000 0100106C-          LDW  *+B14(bits.bit7), A2
13                                ; load field
14 00000004 0109E7A0          AND   0Fh, A2, A2    ; mask off garbage

```

## **.tab**

### **Define Tab Size**

#### **Syntax**

**.tab** *size*

#### **Description**

The **.tab** directive defines the tab size. Tabs encountered in the source input are translated to *size* character spaces in the listing. The default tab size is eight spaces.

#### **Example**

In this example, each of the lines of code following a **.tab** statement consists of a single tab character followed by an NOP instruction.

#### **Source file:**

```
; default tab size
NOP
NOP
NOP
    .tab 4
NOP
NOP
NOP
    .tab 16
NOP
NOP
NOP
```

#### **Listing file:**

```
1                                     ; default tab size
2 00000000 00000000                NOP
3 00000004 00000000                NOP
4 00000008 00000000                NOP
5                                     .tab4
7 0000000c 00000000                NOP
8 00000010 00000000                NOP
9 00000014 00000000                NOP
10                                     .tab 16
12 00000018 00000000                NOP
13 0000001c 00000000                NOP
14 00000020 00000000                NOP
```

**.text****Assemble Into the .text Section****Syntax****.text****Description**

The **.text** directive tells the assembler to begin assembling into the .text section, which usually contains executable code. The section program counter is set to 0 if nothing has yet been assembled into the .text section. If code has already been assembled into the .text section, the section program counter is restored to its previous value in the section.

The .text section is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you use a .data or .sect directive to specify a different section.

For more information about COFF sections, see [Chapter 2, Introduction to Common Object File Format](#).

**Example**

This example assembles code into the .text and .data sections.

```

1          *****
2          ** Begin assembling into .data section. **
3          *****
4 00000000          .data
5 00000000 00000005          .byte  5,6
   00000001 00000006
6
7          *****
8          ** Begin assembling into .text section. **
9          *****
10 00000000          .text
11 00000000 00000001          .byte  1
12 00000001 00000002          .byte  2,3
   00000002 00000003
13
14          *****
15          ** Resume assembling into .data section.**
16          *****
17 00000002          .data
18 00000002 00000007          .byte  7,8
   00000003 00000008
19
20          *****
21          ** Resume assembling into .text section.**
22          *****
23 00000003          .text
24 00000003 00000004          .byte  4

```



<b>.title</b>	<b>Define Page Title</b>
<b>Syntax</b>	<b>.title "string"</b>
<b>Description</b>	<p>The <b>.title</b> directive supplies a title that is printed in the heading on each listing page. The source statement itself is not printed, but the line counter is incremented.</p> <p>The <i>string</i> is a quote-enclosed title of up to 64 characters. If you supply more than 64 characters, the assembler truncates the string and issues a warning:</p> <pre>*** WARNING! line x: W0001: String is too long - will be truncated</pre> <p>The assembler prints the title on the page that follows the directive and on subsequent pages until another <b>.title</b> directive is processed. If you want a title on the first page, the first source statement must contain a <b>.title</b> directive.</p>
<b>Example</b>	<p>In this example, one title is printed on the first page and a different title is printed on succeeding pages.</p> <p><b>Source file:</b></p> <pre>.title  "**** Fast Fourier Transforms ****" ;      . ;      . ;      . .title  "**** Floating-Point Routines ****" .page</pre> <p><b>Listing file:</b></p> <pre>TMS320C6x COFF Assembler   Version x.xx      Tue Apr 14 17:18:21 1997 Copyright ©) 1996-1997 Texas Instruments Incorporated **** Fast Fourier Transforms ****                                PAGE      1        2                      ;      .       3                      ;      .       4                      ;      .  TMS320C6x COFF Assembler   Version x.xx      Tue Apr 14 17:18:21 1997 Copyright ©) 1996-1997 Texas Instruments Incorporated **** Floating-Point Routines ****                                PAGE      2</pre> <p>No Errors, No Warnings</p>

**.union/.endunion/.tag****Declare Union Type****Syntax**

```

[stag]    .union    [expr]
[mem0]  element  [expr0]
[mem1]  element  [expr1]
.
.
.
[memn]  .tag stag  [exprn]
.
.
.
[memN]  element  [exprN]
[size]    .endunion
label     .tag      stag

```

**Description**

The **.union** directive assigns symbolic offsets to the elements of alternate data structure definitions to be allocated in the same memory space. This enables you to define several alternate structures and then let the assembler calculate the element offset. This is similar to a C union. The **.union** directive does not allocate any memory; it merely creates a symbolic template that can be used repeatedly.

A **.struct** definition can contain a **.union** definition, and **.structs** and **.unions** can be nested.

The **.endunion** directive terminates the union definition.

The **.tag** directive gives structure or union characteristics to a *label*, simplifying the symbolic representation and providing the ability to define structures or unions that contain other structures or unions. The **.tag** directive does not allocate memory. The structure or union tag of a **.tag** directive must have been previously defined.

Following are descriptions of the parameters used with the **.struct**, **.endstruct**, and **.tag** directives:

- The *utag* is the union's tag. is the union's tag. Its value is associated with the beginning of the union. If no utag is present, the assembler puts the union members in the global symbol table with the value of their absolute offset from the top of the union. In this case, each member must have a unique name.
- The *expr* is an optional expression indicating the beginning offset of the union. Unions default to start at 0. This parameter can only be used with a top-level union. It cannot be used when defining a nested union.
- The *mem<sub>n/N</sub>* is an optional label for a member of the union. This label is absolute and equates to the present offset from the beginning of the union. A label for a union member cannot be declared global.
- The *element* is one of the following descriptors: **.byte**, **.char**, **.double**, **field**, **.float**, **.half**, **.int**, **.long**, **.short**, **.string**, **.ubyte**, **.uchar**, **.uhalf**, **.uint**, **.ulong**, **.ushort**, **.uword**, and **.word**. An element can also be a complete declaration of a nested structure or union, or a structure or union declared by its tag. Following a **.union** directive, these directives describe the element's size. They do not allocate memory.
- The *expr<sub>n/N</sub>* is an optional expression for the number of elements described. This value defaults to 1. A **.string** element is considered to be one byte in size, and a **.field** element is one bit.
- The *size* is an optional label for the total size of the union.

### Directives That Can Appear in a .union/.endunion Sequence

**Note:** The only directives that can appear in a .union/.endunion sequence are element descriptors, structure and union tags, and conditional assembly directives. Empty structures are illegal.

---

These examples show unions with and without tags.

#### Example 1

```

1                .global employid
2      xample .union                ; utag
3      0000 ival      .word          ; member1 = int
4      0000 fval      .float          ; member2 = float
5      0000 sval      .string         ; member3 = string
6      0002 real_len .endunion      ; real_len = 2
7
8 000000          .bss employid, real_len ;allocate memory
9
10             employid .tag xample    ; name an instance
11 000000 0000-    ADD employid.fval, A ; access union element

```

#### Example 2

```

1
2             .union                ; utag
3      0000 x      .long              ; member1 = long
4      0000 y      .float              ; member2 = float
5      0000 z      .word              ; member3 = word
6      0002 size_u .endunion        ; real_len = 2
7

```

**.usect****Reserve Uninitialized Space****Syntax**

*symbol* **.usect** "section name", size in bytes [, alignment[, bank offset]]

**Description**

The **.usect** directive reserves space for variables in an uninitialized, named section. This directive is similar to the **.bss** directive; both simply reserve space for data and that space has no contents. However, **.usect** defines additional sections that can be placed anywhere in memory, independently of the **.bss** section.

- The *symbol* points to the first location reserved by this invocation of the **.usect** directive. The symbol corresponds to the name of the variable for which you are reserving space.
- The *section name* is significant to 200 characters and must be enclosed in double quotes. This parameter names the uninitialized section. A section name can contain a subsection name in the form *section name:subsection name*.
- The *size in bytes* is an expression that defines the number of bytes that are reserved in *section name*.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. This boundary indicates the size of the slot in bytes and can be set to any power of 2.
- The *bank offset* is an optional parameter that ensures that the space allocated to the symbol occurs on a specific memory bank boundary. The bank offset value measures the number of bytes to offset from the alignment specified before assigning the symbol to that location.

Initialized sections directives (**.text**, **.data**, and **.sect**) end the current section and tell the assembler to begin assembling into another section. A **.usect** or **.bss** directive encountered in the current section is simply assembled, and assembly continues in the current section.

Variables that can be located contiguously in memory can be defined in the same specified section; to do so, repeat the **.usect** directive with the same section name and the subsequent symbol (variable name).

For more information about COFF sections, see [Chapter 2, Introduction to Common Object File Format](#).

**Example**

This example uses the **.usect** directive to define two uninitialized, named sections, **var1** and **var2**. The symbol **ptr** points to the first byte reserved in the **var1** section. The symbol **array** points to the first byte in a block of 100 bytes reserved in **var1**, and **dflag** points to the first byte reserved in the **var2** section.

[Figure 4-8](#) shows how this example reserves space in two uninitialized sections, **var1** and **var2**.

```

1          *****
2          **   Assemble into .text section           **
3          *****
4 00000000          .text
5 00000000 008001A0      MV      A0,A1
6
7          *****
8          **   Reserve 2 bytes in var1.             **
9          *****
10 00000000      ptr .usect "var1",2
11 00000004 0100004C-    LDH      *+B14(ptr),A2    ; still in .text
12
13          *****
14          **   Reserve 100 bytes in var1            **
15          *****
16 00000002      array .usect "var1",100
17 00000008 01800128-    MVK      array,A3          ; still in .text
18 0000000c 01800068-    MVKH     array,A3

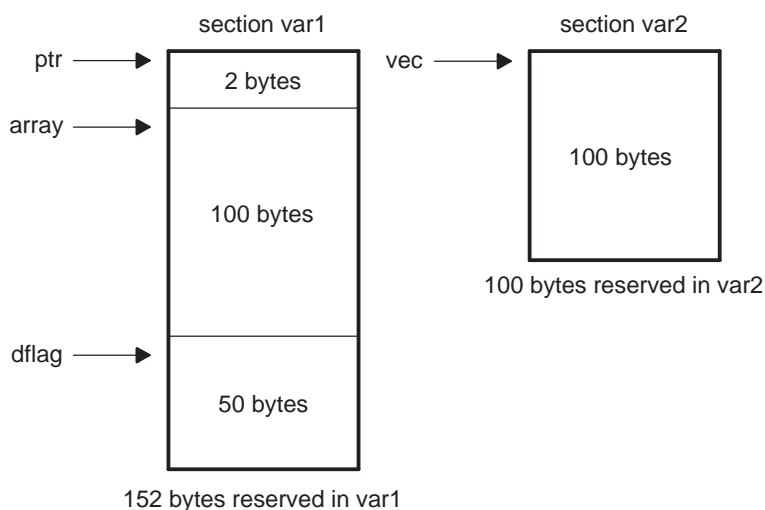
```

```

19
20
21          **** Reserve 50 bytes in var1 ****
22          ****
23 00000066      dflag .usect "var1",50
24 00000010 02003328-      MVK      dflag,A4
25 00000014 02000068-      MVKH     dflag,A4
26
27          ****
28          **** Reserve 100 bytes in var1 ****
29          ****
30 00000000      vec .usect "var2",100
31 00000018 0000002A-      MVK      vec,B0          ; still in .text
32 0000001c 0000006A-      MVKH     vec,B0

```

**Figure 4-8. The .usect Directive**



## **.var**

### **Use Substitution Symbols as Local Variables**

#### **Syntax**

**.var** *sym*<sub>1</sub>[, *sym*<sub>2</sub>, ... , *sym*<sub>*n*</sub>]

#### **Description**

The .var directive allows you to use substitution symbols as local variables within a macro. With this directive, you can define up to 32 local macro substitution symbols (including parameters) per macro.

The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

For more information on macros, see [Chapter 5, Macro Language](#).



## Macro Language

The TMS320C6000™ assembler supports a macro language that enables you to create your own instructions. This is especially useful when a program executes a particular task several times. The macro language lets you:

- Define your own macros and redefine existing macros
- Simplify long or complicated assembly code
- Access macro libraries created with the archiver
- Define conditional and repeatable blocks within a macro
- Manipulate strings within a macro
- Control expansion listing

Topic	Page
<b>5.1 Using Macros .....</b>	<b>128</b>
<b>5.2 Defining Macros.....</b>	<b>128</b>
<b>5.3 Macro Parameters/Substitution Symbols .....</b>	<b>129</b>
<b>5.4 Macro Libraries.....</b>	<b>136</b>
<b>5.5 Using Conditional Assembly in Macros .....</b>	<b>137</b>
<b>5.6 Using Labels in Macros .....</b>	<b>139</b>
<b>5.7 Producing Messages in Macros .....</b>	<b>140</b>
<b>5.8 Using Directives to Format the Output Listing.....</b>	<b>141</b>
<b>5.9 Using Recursive and Nested Macros .....</b>	<b>141</b>
<b>5.10 Macro Directives Summary .....</b>	<b>142</b>

## 5.1 Using Macros

Programs often contain routines that are executed several times. Instead of repeating the source statements for a routine, you can define the routine as a macro, then call the macro in the places where you would normally repeat the routine. This simplifies and shortens your source program.

If you want to call a macro several times but with different data each time, you can assign parameters within a macro. This enables you to pass different information to the macro each time you call it. The macro language supports a special symbol called a *substitution symbol*, which is used for macro parameters. See [Section 5.3, Macro Parameters/Substitution Symbols](#), for more information.

Using a macro is a 3-step process.

**Step 1. Define the macro.** You must define macros before you can use them in your program. There are two methods for defining macros:

- a. Macros can be defined at the beginning of a *source file* or in an *copy/include file*. See [Section 5.2, Defining Macros](#), for more information.
- b. Macros can also be defined in a *macro library*. A macro library is a collection of files in archive format created by the archiver. Each member of the archive file (macro library) may contain one macro definition corresponding to the member name. You can access a macro library by using the `.mlib` directive. For more information, see [Section 5.4, Macro Libraries](#).

**Step 2. Call the macro.** After you have defined a macro, call it by using the macro name as a mnemonic in the source program. This is referred to as a *macro call*.

**Step 3. Expand the macro.** The assembler expands your macros when the source program calls them. During expansion, the assembler passes arguments by variable to the macro parameters, replaces the macro call statement with the macro definition, then assembles the source code. By default, the macro expansions are printed in the listing file. You can turn off expansion listing by using the `.mnoist` directive. For more information, see [Section 5.8, Using Directives to Format the Output Listing](#).

When the assembler encounters a macro definition, it places the macro name in the opcode table. This redefines any previously defined macro, library entry, directive, or instruction mnemonic that has the same name as the macro. This allows you to expand the functions of directives and instructions, as well as to add new instructions.

## 5.2 Defining Macros

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file or in a `.copy/.include` file (see [Copy Source File](#)); they can also be defined in a macro library. For more information, see [Section 5.4, Macro Libraries](#).

Macro definitions can be nested, and they can call other macros, but all elements of the macro must be defined in the same file. Nested macros are discussed in [Section 5.9, Using Recursive and Nested Macros](#).

A macro definition is a series of source statements in the following format:

```

macname .macro [parameter1] [, ... ,parametern]
        model statements or macro directives
        [.mexit]
        .endm
```

*macname* names the macro. You must place the name in the source statement's label field. Only the first 128 characters of a macro name are significant. The assembler places the macro name in the internal opcode table, replacing any instruction or previous macro definition with the same name.



<b>.macro</b>	is the directive that identifies the source statement as the first line of a macro definition. You must place .macro in the opcode field.
<i>parameter<sub>1</sub>,</i> <i>parameter<sub>n</sub></i>	are optional substitution symbols that appear as operands for the .macro directive. Parameters are discussed in <a href="#">Section 5.3, Macro Parameters/Substitution Symbols</a> .
<i>model statements</i>	are instructions or assembler directives that are executed each time the macro is called.
<i>macro directives</i>	are used to control macro expansion.
<b>.mexit</b>	is a directive that functions as a <i>goto .endm</i> . The .mexit directive is useful when error testing confirms that macro expansion fails and completing the rest of the macro is unnecessary.
<b>.endm</b>	is the directive that terminates the macro definition.

[Example 5-1](#) shows the definition, call, and expansion of a macro.

#### Example 5-1. Macro Definition, Call, and Expansion

Macro definition: The following code defines a macro, sadd4, with four parameters:

```

1          sadd4  .macro  r1,r2,r3,r4
2          !
3          !  sadd4  r1, r2 ,r3, r4
4          !  r1 = r1 + r2 + r3 + r4 (saturated)
5          !
6          SADD   r1,r2,r1
7          SADD   r1,r3,r1
8          SADD   r1,r4,r1
9          .endm

```

Macro call: The following code calls the sadd4 macro with four arguments:

```

10
11 00000000          sadd4  A0,A1,A2,A3

```

Macro expansion: The following code shows the substitution of the macro definition for the macro call. The assembler substitutes A0, A1, A2, and A3 for the r1, r2, r3, and r4 parameters of sadd4.

```

1      00000000 00040278      SADD   A0,A1,A0
1      00000004 00080278      SADD   A0,A2,A0
1      00000008 000C0278      SADD   A0,A3,A0

```

If you want to include comments with your macro definition but *do not* want those comments to appear in the macro expansion, use an exclamation point to precede your comments. If you *do* want your comments to appear in the macro expansion, use an asterisk or semicolon. See [Section 5.7, Producing Messages in Macros](#), for more information about macro comments.

## 5.3 Macro Parameters/Substitution Symbols

If you want to call a macro several times with different data each time, you can assign parameters within the macro. The macro language supports a special symbol, called a *substitution symbol*, which is used for macro parameters.

Macro parameters are substitution symbols that represent a character string. These symbols can also be used outside of macros to equate a character string to a symbol name (see [Section 3.8.6, Substitution Symbols](#)).

Valid substitution symbols can be up to 128 characters long and *must begin with a letter*. The remainder of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

## Macro Parameters/Substitution Symbols

Substitution symbols used as macro parameters are local to the macro they are defined in. You can define up to 32 local substitution symbols (including substitution symbols defined with the `.var` directive) per macro. For more information about the `.var` directive, see [Section 5.3.6, Substitution Symbols as Local Variables in Macros](#).

During macro expansion, the assembler passes arguments by variable to the macro parameters. The character-string equivalent of each argument is assigned to the corresponding parameter. Parameters without corresponding arguments are set to the null string. If the number of arguments exceeds the number of parameters, the last parameter is assigned the character-string equivalent of all remaining arguments.

If you pass a list of arguments to one parameter or if you pass a comma or semicolon to a parameter, you must surround these terms with quotation marks.

At assembly time, the assembler replaces the macro parameter/substitution symbol with its corresponding character string, then translates the source code into object code.

[Example 5-2](#) shows the expansion of a macro with varying numbers of arguments.

### Example 5-2. Calling a Macro With Varying Numbers of Arguments

Macro definition:

```

Parms      .macro      a,b,c
;          a = :a:
;          b = :b:
;          c = :c:
          .endm

```

Calling the macro:

<pre> Parms      100,label ;          a = 100 ;          b = label ;          c = " "  Parms      100, , x ;          a = 100 ;          b = " " ;          c = x  Parms      ""string"",x,y ;          a = "string" ;          b = x ;          c = y </pre>	<pre> Parms      100,label,x,y ;          a = 100 ;          b = label ;          c = x,y  Parms      "100,200,300",x,y ;          a = 100,200,300 ;          b = x ;          c = y </pre>
---	---

### 5.3.1 Directives That Define Substitution Symbols

You can manipulate substitution symbols with the `.asg` and `.eval` directives.

- The `.asg` directive assigns a character string to a substitution symbol.

For the `.asg` directive, the quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the *substitution symbol*. The syntax of the `.asg` directive is:

```
.asg["]character string["], substitution symbol
```

[Example 5-3](#) shows character strings being assigned to substitution symbols.

### Example 5-3. The .asg Directive

```
.asg    "A4", RETVAL           ; return value
.asg    "B14", PAGEPTR        ; global page pointer
.asg    ""Version 1.0"", version
.asg    "p1, p2, p3", list
```

- The **.eval** directive performs arithmetic on numeric substitution symbols. The **.eval** directive evaluates the *expression* and assigns the string value of the result to the *substitution symbol*. If the expression is not well defined, the assembler generates an error and assigns the null string to the symbol. The syntax of the **.eval** directive is:

```
.eval well-defined expression , substitution symbol
```

[Example 5-4](#) shows arithmetic being performed on substitution symbols.

### Example 5-4. The .eval Directive

```
.asg    1,counter
.loop   100
.word   counter
.eval   counter + 1,counter
.endloop
```

In [Example 5-4](#), the **.asg** directive could be replaced with the **.eval** directive (**.eval 1, counter**) without changing the output. In simple cases like this, you can use **.eval** and **.asg** interchangeably. However, you must use **.eval** if you want to calculate a *value* from an expression. While **.asg** only assigns a character string to a substitution symbol, **.eval** evaluates an expression and then assigns the character string equivalent to a substitution symbol.

See [Assign a Substitution Symbol](#) for more information about the **.asg** and **.eval** assembler directives.

### 5.3.2 Built-In Substitution Symbol Functions

The following built-in substitution symbol functions enable you to make decisions on the basis of the string value of substitution symbols. These functions always return a value, and they can be used in expressions. Built-in substitution symbol functions are especially useful in conditional assembly expressions. Parameters of these functions are substitution symbols or character-string constants.

In the function definitions shown in [Table 5-1](#), *a* and *b* are parameters that represent substitution symbols or character-string constants. The term *string* refers to the string value of the parameter. The symbol *ch* represents a character constant.

**Table 5-1. Substitution Symbol Functions and Return Values**

Function	Return Value
<b>\$symlen</b> ( <i>a</i> )	Length of string <i>a</i>
<b>\$symcmp</b> ( <i>a,b</i> )	< 0 if <i>a</i> < <i>b</i> ; 0 if <i>a</i> = <i>b</i> ; > 0 if <i>a</i> > <i>b</i>
<b>\$firstch</b> ( <i>a,ch</i> )	Index of the first occurrence of character constant <i>ch</i> in string <i>a</i>
<b>\$lastch</b> ( <i>a,ch</i> )	Index of the last occurrence of character constant <i>ch</i> in string <i>a</i>
<b>\$isdefed</b> ( <i>a</i> )	1 if string <i>a</i> is defined in the symbol table 0 if string <i>a</i> is not defined in the symbol table
<b>\$ismember</b> ( <i>a,b</i> )	Top member of list <i>b</i> is assigned to string <i>a</i> 0 if <i>b</i> is a null string
<b>\$iscons</b> ( <i>a</i> )	1 if string <i>a</i> is a binary constant 2 if string <i>a</i> is an octal constant 3 if string <i>a</i> is a hexadecimal constant 4 if string <i>a</i> is a character constant 5 if string <i>a</i> is a decimal constant
<b>\$isname</b> ( <i>a</i> )	1 if string <i>a</i> is a valid symbol name 0 if string <i>a</i> is not a valid symbol name
<b>\$isreg</b> ( <i>a</i> ) <sup>(1)</sup>	1 if string <i>a</i> is a valid predefined register name 0 if string <i>a</i> is not a valid predefined register name

(1) For more information about predefined register names, see [Section 3.8.5, Predefined Symbolic Constants](#).

[Example 5-5](#) shows built-in substitution symbol functions.

#### **Example 5-5. Using Built-In Substitution Symbol Functions**

```

pushx .macro list
!
! Push more than one item
! $ismember removes the first item in the list

    .var        item
    .loop
    .break      ($ismember(item, list) = 0)
    STW         item,*B15--[1]
    .endloop
    .endm

pushx      A0,A1,A2,A3

```

### 5.3.3 Recursive Substitution Symbols

When the assembler encounters a substitution symbol, it attempts to substitute the corresponding character string. If that string is also a substitution symbol, the assembler performs substitution again. The assembler continues doing this until it encounters a token that is not a substitution symbol or until it encounters a substitution symbol that it has already encountered during this evaluation.

In [Example 5-6](#), the x is substituted for z; z is substituted for y; and y is substituted for x. The assembler recognizes this as infinite recursion and ceases substitution.

#### Example 5-6. Recursive Substitution

```
.asg    "x",z    ; declare z and assign z = "x"
.asg    "z",y    ; declare y and assign y = "z"
.asg    "y",x    ; declare x and assign x = "y"
MVKL    x, A1
MVKH    x, A1

*      MVKL    x, A1    ; recursive expansion
*      MVKH    x, A1    ; recursive expansion
```

### 5.3.4 Forced Substitution

In some cases, substitution symbols are not recognizable to the assembler. The forced substitution operator, which is a set of colons surrounding the symbol, enables you to force the substitution of a symbol's character string. Simply enclose a symbol with colons to force the substitution. Do not include any spaces between the colons and the symbol.

The syntax for the forced substitution operator is:

```
:symbol:
```

The assembler expands substitution symbols surrounded by colons before expanding other substitution symbols.

You can use the forced substitution operator only inside macros, and you cannot nest a forced substitution operator within another forced substitution operator.

[Example 5-7](#) shows how the forced substitution operator is used.

#### Example 5-7. Using the Forced Substitution Operator

```
force    .macro    x
        .loop      8
PORT:x:  .set      x*4
        .eval      x+1, x
        .endloop
        .endm

        .global    portbase
        force

PORT0    .set      0
PORT1    .set      4
.
.
.
PORT7    .set      28
```

### 5.3.5 Accessing Individual Characters of Subscripted Substitution Symbols

In a macro, you can access the individual characters (substrings) of a substitution symbol with subscripted substitution symbols. You must use the forced substitution operator for clarity.

You can access substrings in two ways:

- `:symbol (well-defined expression):`  
This method of subscripting evaluates to a character string with one character.
- `:symbol (well-defined expression1, well-defined expression2):`  
In this method, expression<sub>1</sub> represents the substring's starting position, and expression<sub>2</sub> represents the substring's length. You can specify exactly where to begin subscripting and the exact length of the resulting character string. *The index of substring characters begins with 1, not 0.*

[Example 5-8](#) and [Example 5-9](#) show built-in substitution symbol functions used with subscripted substitution symbols.

#### Example 5-8. Using Subscripted Substitution Symbols to Redefine an Instruction

```
storex .macro      x
      .var        tmp
      .asg        :x(1):, tmp
      .if         $symcmp(tmp,"A") == 0
      STW         x,*A15--(4)
      .elseif     $symcmp(tmp,"B") == 0
      STW         x,*A15--(4)
      .elseif     $iscons(x)
      MVK         x,A0
      STW         A0,*A15--(4)
      .else
      .emsg       "Bad Macro Parameter"
      .endif
      .endm

storex 10h
storex A15
```

In [Example 5-8](#), subscripted substitution symbols redefine the STW instruction so that it handles immediate.

### Example 5-9. Using Subscripted Substitution Symbols to Find Substrings

```

substr .macro    start,strg1,strg2,pos
      .var      len1,len2,i,tmp
      .if       $symlen(start) = 0
      .eval     1,start
      .endif
      .eval     0,pos
      .eval     start,i
      .eval     $symlen(strg1),len1
      .eval     $symlen(strg2),len2
      .loop
      .break    I = (len2 - len1 + 1)
      .asg      ":strg2(i,len1):",tmp
      .if       $symcmp(strg1,tmp) = 0
      .eval     i,pos
      .break
      .else
      .eval     I + 1,i
      .endif
      .endloop
      .endm

      .asg      0,pos
      .asg      "ar1 ar2 ar3 ar4",regs
      substr    1,"ar2",regs,pos
      .word     pos

```

In [Example 5-9](#), the subscripted substitution symbol is used to find a substring strg1 beginning at position start in the string strg2. The position of the substring strg1 is assigned to the substitution symbol pos.

### 5.3.6 Substitution Symbols as Local Variables in Macros

If you want to use substitution symbols as local variables within a macro, you can use the **.var** directive to define up to 32 local macro substitution symbols (including parameters) per macro. The **.var** directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

```
.var sym1 [,sym2 , ... ,symn]
```

The **.var** directive is used in [Example 5-8](#) and [Example 5-9](#).

## 5.4 Macro Libraries

One way to define macros is by creating a macro library. A macro library is a collection of files that contain macro definitions. You must use the archiver to collect these files, or members, into a single file (called an archive). Each member of a macro library contains one macro definition. The files in a macro library must be unassembled source files. The macro name and the member name must be the same, and the macro filename's extension must be .asm. For example:

Macro Name	Filename in Macro Library
simple	simple.asm
add3	add3.asm

You can access the macro library by using the .mlib assembler directive (described in ). The syntax is:

**.mlib** *filename*

When the assembler encounters the .mlib directive, it opens the library named by filename and creates a table of the library's contents. The assembler enters the names of the individual members within the library into the opcode tables as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table.

The assembler expands the library entry in the same way it expands other macros. (See [Section 5.1, Using Macros](#), for how the assembler expands macros.) You can control the listing of library entry expansions with the .mlist directive. For more information about the .mlist directive, see [Section 5.8, Using Directives to Format the Output Listing](#), and [Start/Stop Macro Expansion Listing](#) . Only macros that are actually called from the library are extracted, and they are extracted only once.

You can use the archiver to create a macro library by including the desired files in an archive. A macro library is no different from any other archive, except that the assembler expects the macro library to contain macro definitions. The assembler expects *only* macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable results. For information about creating a macro library archive, see [Chapter 6, Archiver Description](#).



## 5.5 Using Conditional Assembly in Macros

The conditional assembly directives are **.if/.elseif/.else/.endif** and **.loop/.break/.endloop**. They can be nested within each other up to 32 levels deep. The format of a conditional block is:

```
.if well-defined expression
[.elseif well-defined expression]
[.else]
.endif
```

The **.elseif** and **.else** directives are optional in conditional assembly. The **.elseif** directive can be used more than once within a conditional assembly code block. When **.elseif** and **.else** are omitted and when the **.if** expression is false (0), the assembler continues to the code following the **.endif** directive. See [Assemble Conditional Blocks](#) for more information on the **.if/.elseif/.else/.endif** directives.

The **.loop/.break/.endloop** directives enable you to assemble a code block repeatedly. The format of a repeatable block is:

```
.loop [well-defined expression]
[.break [well-defined expression]]
.endloop
```

The **.loop** directive's optional *well-defined expression* evaluates to the loop count (the number of loops to be performed). If the expression is omitted, the loop count defaults to 1024 unless the assembler encounters a **.break** directive with an expression that is true (nonzero). See [Assemble Code Block Repeatedly](#) for more information on the **.loop/.break/.endloop** directives.

The **.break** directive and its expression are optional in repetitive assembly. If the expression evaluates to false, the loop continues. The assembler breaks the loop when the **.break** expression evaluates to true or when the **.break** expression is omitted. When the loop is broken, the assembler continues with the code after the **.endloop** directive.

[Example 5-10](#), [Example 5-11](#), and [Example 5-12](#) show the **.loop/.break/.endloop** directives, properly nested conditional assembly directives, and built-in substitution symbol functions used in a conditional assembly code block.

### Example 5-10. The .loop/.break/.endloop Directives

```
.asg      1,x
.loop

.break   (x == 10)      ; if x == 10, quit loop/break with expression

.eval    x+1,x
.endloop
```

**Example 5-11. Nested Conditional Assembly Directives**

```

.asg      1,x
.loop

.if (x == 10)      ; if x == 10, quit loop
.break (x == 10)   ; force break
.endif

.eval    x+1,x
.endloop

```

**Example 5-12. Built-In Substitution Symbol Functions in a Conditional Assembly Code Block**

```

MACK3    .macro src1, src2, sum, k
!
!    dst = dst + k * (src1 * src2)

        .if      k = 0
MPY      src1, src2, src2
NOP
ADD      src2, sum, sum
        .else
MPY      src1,src2,src2
MVK      k,src1
MPY      src1,src2,src2
NOP
ADD      src2,sum,sum
        .endif

        .endm

MACK3    A0,A1,A3,0
MACK3    A0,A1,A3,100

```

For more information, see [Section 4.7, Directives That Enable Conditional Assembly](#).

## 5.6 Using Labels in Macros

All labels in an assembly language program must be unique. This includes labels in macros. If a macro is expanded more than once, its labels are defined more than once. *Defining a label more than once is illegal.* The macro language provides a method of defining labels in macros so that the labels are unique. Simply follow each label with a question mark, and the assembler replaces the question mark with a period followed by a unique number. When the macro is expanded, *you do not see the unique number in the listing file.* Your label appears with the question mark as it did in the macro definition. You cannot declare this label as global. The syntax for a unique label is:

label ?

Example 5-13 shows unique label generation in a macro.

### Example 5-13. Unique Labels in a Macro

1	min	.macro	x,y,z
2			
3		MV	y,z
4		CMPLT	x,y,y
5	[y]	B	l?
6		NOP	5
7		MV	x,z
8	l?		
9		.endm	
10			
11			
12	00000000	MIN	A0,A1,A2
1			
1	00000000 010401A1	MV	A1,A2
1	00000004 00840AF8	CMPLT	A0,A1,A1
1	00000008 80000292 [A1]	B	l?
1	0000000c 00008000	NOP	5
1	00000010 010001A0	MV	A0,A2
1	00000014	l?	
LABEL	VALUE	DEFN	REF
.TMS320C60	00000001	0	
.tms320C60	00000001	0	
l\$1\$	00000014'	12	12

The maximum label length is shortened to allow for the unique suffix. For example, if the macro is expanded fewer than 10 times, the maximum label length is 126 characters. If the macro is expanded from 10 to 99 times, the maximum label length is 125. The label with its unique suffix is shown in the cross-listing file. To obtain a cross-listing file, invoke the assembler with the -ax option (see [Section 3.3](#)).

## 5.7 Producing Messages in Macros

The macro language supports three directives that enable you to define your own assembly-time error and warning messages. These directives are especially useful when you want to create messages specific to your needs. The last line of the listing file shows the error and warning counts. These counts alert you to problems in your code and are especially useful during debugging.

- .emsg** sends error messages to the listing file. The .emsg directive generates errors in the same manner as the assembler, incrementing the error count and preventing the assembler from producing an object file.
- .mmsg** sends assembly-time messages to the listing file. The .mmsg directive functions in the same manner as the .emsg directive but does not set the error count or prevent the creation of an object file.
- .wmsg** sends warning messages to the listing file. The .wmsg directive functions in the same manner as the .emsg directive, but it increments the warning count and does not prevent the generation of an object file.

**Macro comments** are comments that appear in the definition of the macro *but do not show up in the expansion of the macro*. An exclamation point in column 1 identifies a macro comment. If you want your comments to appear in the macro expansion, precede your comment with an asterisk or semicolon.

[Example 5-14](#) shows user messages in macros and macro comments that do not appear in the macro expansion.

### Example 5-14. Producing Messages in a Macro

```
TEST    .macro    x,y
!
! This macro checks for the correct number of parameters.
! It generates an error message if x and y are not present.
!
! The first line tests for proper input.
!
    .if        ($symlen(x) + ||$symlen(y) == 0)
    .emsg      "ERROR --missing parameter in call to TEST"
    .mexit
    .else
        .
        .
    .endif
    .if
        .
        .
    .endif
    .endm
```

For more information about the .emsg, .mmsg, and .wmsg assembler directives, see [Define Messages](#).

## 5.8 Using Directives to Format the Output Listing

Macros, substitution symbols, and conditional assembly directives may hide information. You may need to see this hidden information, so the macro language supports an expanded listing capability.

By default, the assembler shows macro expansions and false conditional blocks in the list output file. You may want to turn this listing off or on within your listing file. Four sets of directives enable you to control the listing of this information:

- **Macro and loop expansion listing**

**.mlist** expands macros and .loop/.endloop blocks. The .mlist directive prints all code encountered in those blocks.

**.mnolist** suppresses the listing of macro expansions and .loop/ .endloop blocks.

For macro and loop expansion listing, .mlist is the default.

- **False conditional block listing**

**.fclist** causes the assembler to include in the listing file all conditional blocks that do not generate code (false conditional blocks). Conditional blocks appear in the listing exactly as they appear in the source code.

**.fcnolist** suppresses the listing of false conditional blocks. Only the code in conditional blocks that actually assemble appears in the listing. The .if, .elseif, .else, and .endif directives do not appear in the listing.

For false conditional block listing, .fclist is the default.

- **Substitution symbol expansion listing**

**.sslist** expands substitution symbols in the listing. This is useful for debugging the expansion of substitution symbols. The expanded line appears below the actual source line.

**.ssnolist** turns off substitution symbol expansion in the listing.

For substitution symbol expansion listing, .ssnolist is the default.

- **Directive listing**

**.drlist** causes the assembler to print to the listing file all directive lines.

**.drnolist** suppresses the printing of certain directives in the listing file. These directives are .asg, .eval, .var, .sslist, .mlist, .fclist, .ssnolist, .mnolist, .fcnolist, .emsg, .wmsg, .mmsg, .length, .width, and .break.

For directive listing, .drlist is the default.

## 5.9 Using Recursive and Nested Macros

The macro language supports recursive and nested macro calls. This means that you can call other macros in a macro definition. You can nest macros up to 32 levels deep. When you use recursive macros, you call a macro from its own definition (the macro calls itself).

When you create recursive or nested macros, you should pay close attention to the arguments that you pass to macro parameters because the assembler uses dynamic scoping for parameters. This means that the called macro uses the environment of the macro from which it was called.

[Example 5-15](#) shows nested macros. The y in the in\_block macro hides the y in the out\_block macro. The x and z from the out\_block macro, however, are accessible to the in\_block macro.

**Example 5-15. Using Nested Macros**

```

in_block .macro y,a
    .          ; visible parameters are y,a and x,z
    .          ;      from the calling macro
    .endm

out_block .macro    x,y,z
    .          ; visible parameters are x,y,z
    .
    in_block x,y  ; macro call with x and y as arguments
    .
    .endm
    out_block    ; macro call

```

[Example 5-16](#) shows recursive and fact macros. The fact macro produces assembly code necessary to calculate the factorial of n, where n is an immediate value. The result is placed in the A1 register. The fact macro accomplishes this by calling fact1, which calls itself recursively.

**Example 5-16. Using Recursive Macros**

```

.fcno1ist

fact1 .macro n

    .if n == 1
        MVK globcnt, A1          ; Leave the answer in the A1 register.
    .else
        .eval 1, temp            ; Compute the decrement of symbol n.
        .eval globcnt*temp, globcnt ; Multiply to get a new result.
        fact1 temp              ; Recursive call.
    .endif
    .endm

fact .macro n
    .if ! $iscons(n)             ; Test that input is a constant.
        .emsg "Parm not a constant"

    .elseif n < 1                 ; Type check input.
        MVK 0, A1

    .else
        .var temp
        .asg n, globcnt

        fact1 n                  ; Perform recursive procedure

    .endif
    .endm

```

**5.10 Macro Directives Summary**

The directives listed in [Table 5-2](#) through [Table 5-6](#) can be used with macros. The .macro, .mexit, .endm and .var directives are valid only with macros; the remaining directives are general assembly language directives.

**Table 5-2. Creating Macros**

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
<b>.endm</b>	End macro definition	<a href="#">Section 5.2</a>	<a href="#">.endm</a>
<b>macname .macro</b> [ <i>parameter</i> <sub>1</sub> ][ <i>...</i> , <i>parameter</i> <sub>n</sub> ]	Define macro by <i>macname</i>	<a href="#">Section 5.2</a>	<a href="#">.macro</a>
<b>.mexit</b>	Go to .endm	<a href="#">Section 5.2</a>	<a href="#">Section 5.2</a>
<b>.mlib</b> <i>filename</i>	Identify library containing macro definitions	<a href="#">Section 5.4</a>	<a href="#">.mlib</a>

**Table 5-3. Manipulating Substitution Symbols**

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
<b>.asg</b> [ <i>character string</i> ["], <i>substitution symbol</i>	Assign character string to substitution symbol	<a href="#">Section 5.3.1</a>	<a href="#">.asg</a>
<b>.eval</b> <i>well-defined expression</i> , <i>substitution symbol</i>	Perform arithmetic on numeric substitution symbols	<a href="#">Section 5.3.1</a>	<a href="#">.eval</a>
<b>.var</b> <i>sym</i> <sub>1</sub> [ <i>sym</i> <sub>2</sub> , ..., <i>sym</i> <sub>n</sub> ]	Define local macro symbols	<a href="#">Section 5.3.6</a>	<a href="#">.var</a>

**Table 5-4. Conditional Assembly**

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
<b>.break</b> [ <i>well-defined expression</i> ]	Optional repeatable block assembly	<a href="#">Section 5.5</a>	<a href="#">.break</a>
<b>.endif</b>	End conditional assembly	<a href="#">Section 5.5</a>	<a href="#">.endif</a>
<b>.endloop</b>	End repeatable block assembly	<a href="#">Section 5.5</a>	<a href="#">.endloop</a>
<b>.else</b>	Optional conditional assembly block	<a href="#">Section 5.5</a>	<a href="#">.else</a>
<b>.elseif</b> <i>well-defined expression</i>	Optional conditional assembly block	<a href="#">Section 5.5</a>	<a href="#">.elseif</a>
<b>.if</b> <i>well-defined expression</i>	Begin conditional assembly	<a href="#">Section 5.5</a>	<a href="#">.if</a>
<b>.loop</b> [ <i>well-defined expression</i> ]	Begin repeatable block assembly	<a href="#">Section 5.5</a>	<a href="#">.loop</a>

**Table 5-5. Producing Assembly-Time Messages**

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
<b>.emsg</b>	Send error message to standard output	<a href="#">Section 5.7</a>	<a href="#">.emsg</a>
<b>.mmsg</b>	Send assembly-time message to standard output	<a href="#">Section 5.7</a>	<a href="#">.mmsg</a>
<b>.wmsg</b>	Send warning message to standard output	<a href="#">Section 5.7</a>	<a href="#">.wmsg</a>

**Table 5-6. Formatting the Listing**

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
<b>.fclist</b>	Allow false conditional code block listing (default)	<a href="#">Section 5.8</a>	<a href="#">.fclist</a>
<b>.fcnolist</b>	Suppress false conditional code block listing	<a href="#">Section 5.8</a>	<a href="#">.fcnolist</a>
<b>.mlist</b>	Allow macro listings (default)	<a href="#">Section 5.8</a>	<a href="#">.mlist</a>
<b>.mnolist</b>	Suppress macro listings	<a href="#">Section 5.8</a>	<a href="#">.mnolist</a>
<b>.sslist</b>	Allow expanded substitution symbol listing	<a href="#">Section 5.8</a>	<a href="#">.sslist</a>
<b>.ssnolist</b>	Suppress expanded substitution symbol listing (default)	<a href="#">Section 5.8</a>	<a href="#">.ssnolist</a>





## **Archiver Description**

---

---

---

The TMS320C6000™ archiver lets you combine several individual files into a single archive file. For example, you can collect several macros into a macro library. The assembler searches the library and uses the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members.

Topic	Page
<b>6.1 Archiver Overview.....</b>	<b>146</b>
<b>6.2 The Archiver's Role in the Software Development Flow .....</b>	<b>147</b>
<b>6.3 Invoking the Archiver .....</b>	<b>148</b>
<b>6.4 Archiver Examples .....</b>	<b>149</b>

## 6.1 Archiver Overview

You can build libraries from any type of files. Both the assembler and the linker accept archive libraries as input; the assembler can use libraries that contain individual source files, and the linker can use libraries that contain individual object files.

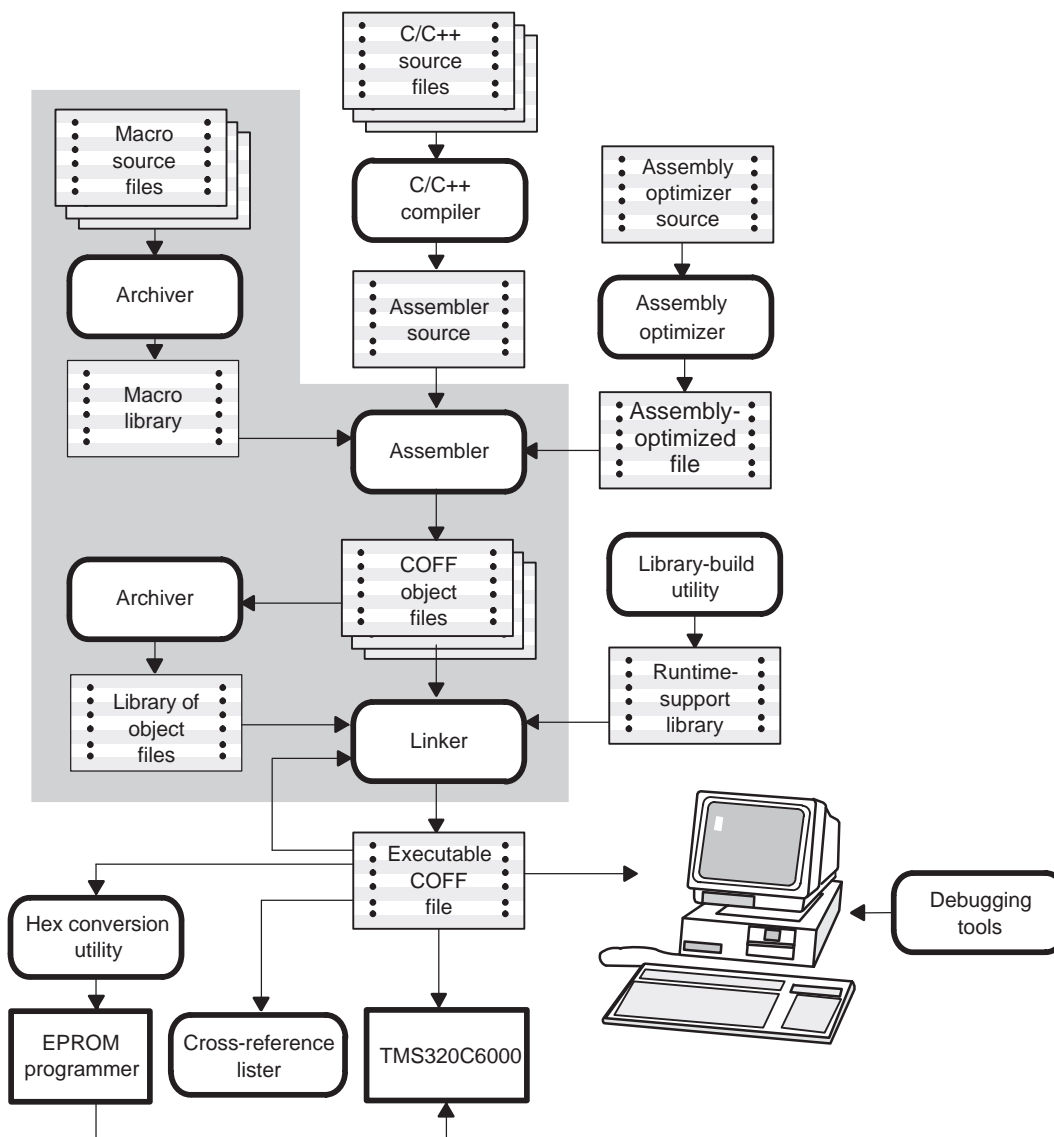
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The linker searches the library and includes members that resolve external references.

You can also use the archiver to build macro libraries. You can create several source files, each of which contains a single macro, and use the archiver to collect these macros into a single, functional group. You can use the `.mlib` directive during assembly to specify that macro library to be searched for the macros that you call. [Chapter 5, \*Macro Language\*](#), discusses macros and macro libraries in detail, while this chapter explains how to use the archiver to build libraries.

## 6.2 The Archiver's Role in the Software Development Flow

Figure 6-1 shows the archiver's role in the software development process. The shaded portion highlights the most common archiver development path. Both the assembler and the linker accept libraries as input.

Figure 6-1. The Archiver in the TMS320C6000 Software Development Flow



### 6.3 Invoking the Archiver

To invoke the archiver, enter:

**ar6x** [-]command [options] libname [filename<sub>1</sub> ... filename<sub>n</sub>]

**ar6x** is the command that invokes the archiver.

**[-]command** tells the archiver how to manipulate the existing library members and any specified . A command can be preceded by an optional hyphen. You must use one of the following commands when you invoke the archiver, but you can use only one command per invocation. The archiver commands are as follows:

- @** uses the contents of the specified file instead of command line entries. You can use this command to avoid limitations on command line length imposed by the host operating system. Use a ; at the beginning of a line in the command file to include comments. (See [Example 6-1](#) for an example using an archiver command file.)
- a** adds the specified files to the library. This command does not replace an existing member that has the same name as an added file; it simply *appends* new members to the end of the archive.
- d** deletes the specified members from the library.
- r** replaces the specified members in the library. If you do not specify filenames, the archiver replaces the library members with files of the same name in the current directory. If the specified file is not found in the library, the archiver adds it instead of replacing it.
- t** prints a table of contents of the library. If you specify filenames, only those files are listed. If you do not specify any filenames, the archiver lists all the members in the specified library.
- x** extracts the specified files. If you do not specify member names, the archiver extracts all library members. When the archiver extracts a member, it simply copies the member into the current directory; it *does not* remove it from the library.

**options** In addition to one of the *commands*, you can specify options. To use options, combine them with a command; for example, to use the a command and the s option, enter -as or as. The hyphen is optional for archiver options only. These are the archiver options:

- q** (quiet) suppresses the banner and status messages.
- s** prints a list of the global symbols that are defined in the library. (This option is valid only with the a, r, and d commands.)
- u** replaces library members only if the replacement has a more recent modification date. You must use the r command with the -u option to specify which members to replace.
- v** (verbose) provides a file-by-file description of the creation of a new library from an old library and its members.

**libname** names the archive library to be built or modified. If you do not specify an extension for *libname*, the archiver uses the default extension *.lib*.

**filenames** names individual files to be manipulated. These files can be existing library members or new files to be added to the library. When you enter a filename, you must enter a complete filename including extension, if applicable.

### Naming Library Members

- Note:** It is possible (but not desirable) for a library to contain several members with the same name. If you attempt to delete, replace, or extract a member whose name is the same as another library member, the archiver deletes, replaces, or extracts the first library member with that name.

## 6.4 Archiver Examples

The following are examples of typical archiver operations:

- If you want to create a library called function.lib that contains the files sine.obj, cos.obj, and flt.obj, enter:

```
ar6x -a function sine.obj cos.obj flt.obj
```

The archiver responds as follows:

```
==> new archive 'function.lib'
==> building new archive 'function.lib'
```

- You can print a table of contents of function.lib with the -t command, enter:

```
ar6x -t function
```

The archiver responds as follows:

FILE NAME	SIZE	DATE
sine.obj	300	Wed Apr 16 10:00:24 1997
cos.obj	300	Wed Apr 16 10:00:30 1997
flt.obj	300	Wed Apr 16 09:59:56 1997

- If you want to add new members to the library, enter:

```
ar6x -as function atan.obj
```

The archiver responds as follows:

```
==> symbol defined: '_sin'
==> symbol defined: '$sin'
==> symbol defined: '_cos'
==> symbol defined: '$cos'
==> symbol defined: '_tan'
==> symbol defined: '$tan'
==> symbol defined: '_atan'
==> symbol defined: '$atan'
==> building archive 'function.lib'
```

Because this example does not specify an extension for the libname, the archiver adds the files to the library called function.lib. If function.lib does not exist, the archiver creates it. (The -s option tells the archiver to list the global symbols that are defined in the library.)

- If you want to modify a library member, you can extract it, edit it, and replace it. In this example, assume there is a library named macros.lib that contains the members push.asm, pop.asm, and swap.asm.

```
ar6x -x macros push.asm
```

The archiver makes a copy of push.asm and places it in the current directory; it does not remove push.asm from the library. Now you can edit the extracted file. To replace the copy of push.asm in the library with the edited copy, enter:

```
ar6x -r macros push.asm
```

## Archiver Examples

---

- If you want to use a command file, specify the command filename after the -@ command. For example:

```
ar6x -@modules.cmd
```

The archiver responds as follows:

```
==> building archive 'modules.lib'
```

**Example 6-1** is the modules.cmd command file. The r command specifies that the filenames given in the command file replace files of the same name in the modules.lib library. The -u option specifies that these files are replaced only when the current file has a more recent revision date than the file that is in the library.

### Example 6-1. Archiver Command File

```
; Command file to replace members of the
;   modules library with updated files
; Use r command and u option:
ru
; Specify library name:
modules.lib
; List filenames to be replaced if updated:
align.asm
bss.asm
data.asm
text.asm
sect.asm
clink.asm
copy.asm
double.asm
drnolist.asm
emsg.asm
end.asm
```

## Linker Description

The TMS320C6000™ linker creates executable modules by combining COFF object files. This chapter describes the linker options, directives, and statements used to create executable modules. Object libraries, command files, and other key concepts are discussed as well.

The concept of COFF sections is basic to linker operation; [Chapter 2, Introduction to Common Object File Format](#), discusses the COFF format in detail.

Topic	Page
<b>7.1 Linker Overview .....</b>	<b>152</b>
<b>7.2 The Linker's Role in the Software Development Flow .....</b>	<b>153</b>
<b>7.3 Invoking the Linker .....</b>	<b>154</b>
<b>7.4 Linker Options .....</b>	<b>155</b>
<b>7.5 Linker Command Files .....</b>	<b>166</b>
<b>7.6 Object Libraries .....</b>	<b>168</b>
<b>7.7 The MEMORY Directive .....</b>	<b>169</b>
<b>7.8 The SECTIONS Directive .....</b>	<b>171</b>
<b>7.9 Specifying a Section's Run-Time Address .....</b>	<b>182</b>
<b>7.10 Using UNION and GROUP Statements .....</b>	<b>185</b>
<b>7.11 Special Section Types (DSECT, COPY, and NOLOAD) .....</b>	<b>188</b>
<b>7.12 Default Allocation Algorithm .....</b>	<b>188</b>
<b>7.13 Assigning Symbols at Link Time .....</b>	<b>190</b>
<b>7.14 Creating and Filling Holes .....</b>	<b>196</b>
<b>7.15 Linker-Generated Copy Tables .....</b>	<b>198</b>
<b>7.16 Partial (Incremental) Linking .....</b>	<b>207</b>
<b>7.17 Linking C/C++ Code .....</b>	<b>208</b>
<b>7.18 Linker Example .....</b>	<b>210</b>

## 7.1 Linker Overview

The TMS320C6000 linker allows you to configure system memory by allocating output sections efficiently into the memory map. As the linker combines object files, it performs the following tasks:

- Allocates sections into the target system's configured memory
- Relocates symbols and sections to assign them to final addresses
- Resolves undefined external references between input files

The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow you to:

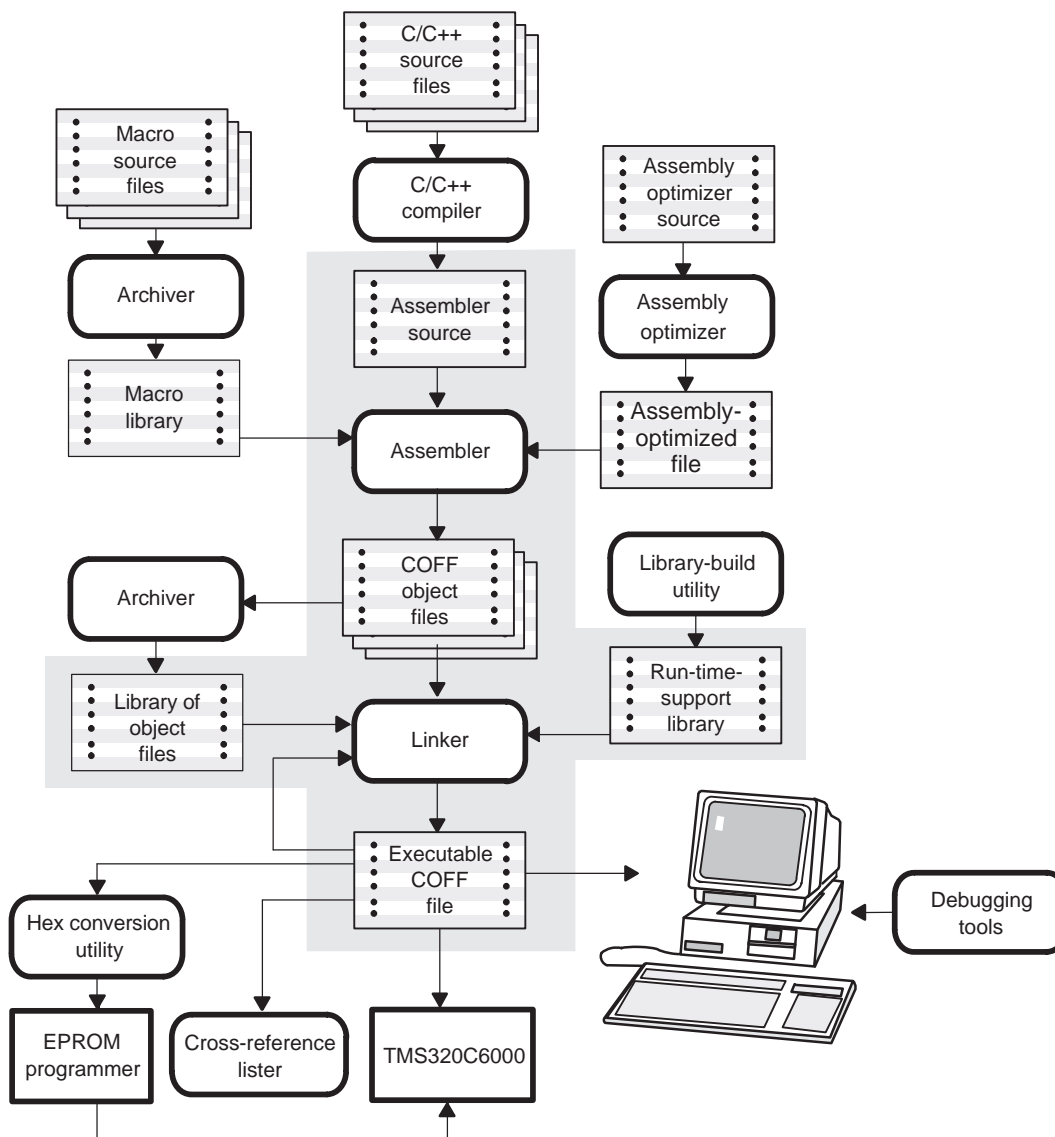
- Allocate sections into specific areas of memory
- Combine object file sections
- Define or redefine global symbols at link time



## 7.2 The Linker's Role in the Software Development Flow

Figure 7-1 illustrates the linker's role in the software development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable COFF object module that can be downloaded to one of several development tools or executed by a TMS320C6000 device.

**Figure 7-1. The Linker in the TMS320C6000 Software Development Flow**



### 7.3 Invoking the Linker

The general syntax for invoking the linker is:

```
cl6x -z [options] filename1 .... filenamen
```

<b>cl6x -z</b>	is the command that invokes the linker.
<i>options</i>	can appear anywhere on the command line or in a linker command file. (Options are discussed in <a href="#">Section 7.4, Linker Options</a> .)
<i>filename<sub>1</sub></i> , <i>filename<sub>n</sub></i>	can be object files, linker command files, or archive libraries. The default extension for all input files is <i>.obj</i> ; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is <i>a.out</i> , unless you use the <i>-o</i> option to name the output file.

There are two methods for invoking the linker:

- Specify options and filenames on the command line. This example links two files, *file1.obj* and *file2.obj*, and creates an output module named *link.out*.

```
cl6x -z file1.obj file2.obj -o link.out
```

- Put filenames and options in a linker command file. Filenames that are specified inside a linker command file must begin with a letter. For example, assume the file *linker.cmd* contains the following lines:

```
-o link.out
file1.obj
file2.obj
```

Now you can invoke the linker from the command line; specify the command filename as an input file:

```
cl6x -z linker.cmd
```

When you use a command file, you can also specify other options and files on the command line. For example, you could enter:

```
cl6x -z -m link.map linker.cmd file3.obj
```

The linker reads and processes a command file as soon as it encounters the filename on the command line, so it links the files in this order: *file1.obj*, *file2.obj*, and *file3.obj*. This example creates an output file called *link.out* and a map file called *link.map*.

For information on invoking the linker for C/C++ files, see [Section 7.17, Linking C/C++ Code](#).

## 7.4 Linker Options

Linker options control linking operations. They can be placed on the command line or in a command file. Linker options must be preceded by a hyphen (-). Options can be separated from arguments (if they have them) by an optional space. [Table 7-1](#) summarizes the linker options.

**Table 7-1. Linker Options Summary**

Option	Description	Section
<b>-a</b>	Produces an absolute, executable module. This is the default; if neither -a nor -r is specified, the linker acts as if -a were specified.	<a href="#">Section 7.4.1.1</a>
<b>-abs</b>	Produces an absolute listing file	<a href="#">Section 7.4.2</a>
<b>-ar</b>	Produces a relocatable, executable object module	<a href="#">Section 7.4.1.3</a>
<b>--args</b>	Allocates memory to be used by the loader to pass arguments	<a href="#">Section 7.4.3</a>
<b>-b</b>	Disables merge of symbolic debugging information	<a href="#">Section 7.4.4</a>
<b>-c</b>	Autoinitializes variables at run time	<a href="#">Section 7.4.5</a>
<b>-cr</b>	Initializes variables at load time	<a href="#">Section 7.4.5</a>
<b>-e=global_symbol</b>	Defines a global symbol that specifies the primary entry point for the output module	<a href="#">Section 7.4.6</a>
<b>-f=fill_value</b>	Sets default fill values for holes within output sections; <i>fill_value</i> is a 32-bit constant	<a href="#">Section 7.4.7</a>
<b>-g=symbol</b>	Makes <i>symbol</i> global (overrides -h)	<a href="#">Section 7.4.8</a>
<b>-h</b>	Makes all global symbols static	<a href="#">Section 7.4.9</a>
<b>-heap= size</b>	Sets heap size (for the dynamic memory allocation in C) to <i>size</i> words and defines a global symbol that specifies the heap size. Default = 1K bytes	<a href="#">Section 7.4.10</a>
<b>-help</b> or <b>-?</b>	Displays information about syntax and available options	—
<b>-l=pathname</b>	Alters library-search algorithms to look in a directory named with <i>pathname</i> before looking in the default location. This option must appear before the -l option.	<a href="#">Section 7.4.11.1</a>
<b>-j</b>	Disables conditional linking	<a href="#">Section 7.4.12</a>
<b>-l= filename</b>	Names an archive library or linker command <i>filename</i> as linker input	<a href="#">Section 7.4.11</a>
<b>-m=filename</b>	Produces a map or listing of the input and output sections, including holes, and places the listing in <i>filename</i>	<a href="#">Section 7.4.13</a>
<b>-o= filename</b>	Names the executable output module. The default filename is a.out.	
<b>-priority</b>	Satisfies unresolved references by the first library that contains a definition for that symbol	<a href="#">Section 7.4.20</a>
<b>-r</b>	Produces a nonexecutable, relocatable output module	<a href="#">Section 7.4.1.2</a>
<b>-s</b>	Strips symbol table information and line number entries from the output module	<a href="#">Section 7.4.15</a>
<b>-stack= size</b>	Sets C system stack size to <i>size</i> words and defines a global symbol that specifies the stack size. Default = 1K bytes	<a href="#">Section 7.4.16</a>
<b>--trampolines</b>	Generates far call trampolines	<a href="#">Section 7.4.17</a>
<b>-u=symbol</b>	Places an unresolved external <i>symbol</i> into the output module's symbol table	<a href="#">Section 7.4.18</a>
<b>-w</b>	Displays a message when an undefined output section is created	<a href="#">Section 7.4.19</a>
<b>-x</b>	Forces rereading of libraries, which resolves back references	<a href="#">Section 7.4.20</a>
<b>--xml_link_info file</b>	Generates a well-formed XML <i>file</i> containing detailed information about the result of a link	<a href="#">Section 7.4.21</a>

### 7.4.1 Relocation Capabilities (-a and -r Options)

The linker performs relocation, which is the process of adjusting all references to a symbol when the symbol's address changes. The linker supports two options (-a and -r) that allow you to produce an absolute or a relocatable output module.

When the linker encounters a file that contains no relocation or symbol table information, it issues a warning message (but continues executing). Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated (that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

#### 7.4.1.1 Producing an absolute output module (-a option)

When you use the -a option without the -r option, the linker produces an *absolute, executable* output module. Absolute files contain *no* relocation information. Executable files contain the following:

- Special symbols defined by the linker (see [Section 7.13.4](#))
- An optional header that describes information such as the program entry point
- No unresolved references

The following example links file1.obj and file2.obj and creates an absolute output module called a.out:

```
cl6x -z -a file1.obj file2.obj
```

---

#### The -a and -r Options

**Note:** If you do not use the -a or the -r option, the linker acts as if you specified -a.

---

#### 7.4.1.2 Producing a relocatable output module (-r option)

When you use the -r option without the -a option, the linker retains relocation entries in the output module. If the output module is relocated (at load time) or relinked (by another linker execution), use -r to retain the relocation entries.

The linker produces a file that is not executable when you use the -r option without -a. A file that is not executable does not contain special linker symbols or an optional header. The file can contain unresolved references, but these references do not prevent creation of an output module.

This example links file1.obj and file2.obj and creates a relocatable output module called a.out:

```
cl6x -z -r file1.obj file2.obj
```

The output file a.out can be relinked with other object files or relocated at load time. (Linking a file that will be relinked with other files is called partial linking. For more information, see [Section 7.16, Partial \(Incremental\) Linking](#).)

#### 7.4.1.3 Producing an executable relocatable output module (-ar option combination)

If you invoke the linker with both the -a and -r options, the linker produces an *executable, relocatable* object module. The output file contains the special linker symbols, an optional header, and all resolved symbol references; however, the relocation information is retained.

This example links file1.obj and file2.obj and creates an executable, relocatable output module called xr.out:

```
cl6x -z -ar file1.obj file2.obj -o xr.out
```

### 7.4.2 Create an Absolute Listing File (-abs Option)

The -abs option produces an output file for each file that was linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

### 7.4.3 Allocate Memory for Use by the Loader to Pass Arguments (--args Option)

The --args option instructs the linker to allocate memory to be used by the loader to pass arguments from the command line of the loader to the program. The syntax of the --args option is:

**-args = size**

The *size* is a number representing the number of bytes to be allocated in target memory for command-line arguments.

By default, the linker creates the `__c_args__` symbol and sets it to -1. When you specify --args=size, the following occur:

- The linker creates an uninitialized section named `.args` of *size* bytes.
- The `__c_args__` symbol contains the address of the `.args` section.

The loader and the target boot code use the `.args` section and the `__c_args__` symbol to determine whether and how to pass arguments from the host to the target program. See the *TMS320C6000 Optimizing Compiler User's Guide* for information about the loader.

### 7.4.4 Disable Merge of Symbolic Debugging Information (-b Option)

By default, the linker eliminates duplicate entries of symbolic debugging information. Such duplicate information is commonly generated when a C program is compiled for debugging. For example:

```
-[ header.h ]-
typedef struct
{
    <define some structure members>
} XYZ;

-[ f1.c ]-
#include "header.h"
...

-[ f2.c ]-
#include "header.h"
...
```

When these files are compiled for debugging, both `f1.obj` and `f2.obj` have symbolic debugging entries to describe type `XYZ`. For the final output file, only one set of these entries is necessary. The linker eliminates the duplicate entries automatically.

Use the -b option if you want the linker to keep such duplicate entries. Using the -b option has the effect of the linker running faster and using less machine memory.

### 7.4.5 C Language Options (-c and -cr Options)

The -c and -cr options cause the linker to use linking conventions that are required by the C compiler.

- The -c option tells the linker to autoinitialize variables at run time.
- The -cr option tells the linker to initialize variables at load time.

For more information, see [Section 7.17, Linking C Code](#), [Section 7.17.4, Autoinitialization of Variables at Run Time](#), and [Section 7.17.5, Initialization of Variables at Load Time](#).

### 7.4.6 Define an Entry Point (-e global\_symbol Option)

The memory address at which a program begins executing is called the *entry point*. When a loader loads a program into target memory, the program counter (PC) must be initialized to the entry point; the PC then points to the beginning of the program.

## Linker Options

The linker can assign one of four values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table.

- The value specified by the `-e` option. The syntax is:

```
-e= global_symbol
```

where *global\_symbol* defines the entry point and must be as an external symbol of the input files.

- The value of symbol `_c_int00` (if present). The `_c_int00` symbol *must* be the entry point if you are linking code produced by the C compiler.
- The value of symbol `_main` (if present)
- 0 (default value)

This example links `file1.obj` and `file2.obj`. The symbol `begin` is the entry point; `begin` must be defined as external in `file1` or `file2`.

```
cl6x -z -e=begin file1.obj file2.obj
```

### 7.4.7 Set Default Fill Value (`-f fill_value` Option)

The `-f` option fills the holes formed within output sections. The syntax for the `-f` option is:

```
-f= fill_value
```

The argument *fill\_value* is a 32-bit constant (up to eight hexadecimal digits). If you do not use `-f`, the linker uses 0 as the default fill value.

This example fills holes with the hexadecimal value `ABCDABCD`:

```
cl6x -z -f=0xABCDABCD file1.obj file2.obj
```

### 7.4.8 Make a Symbol Global (`-g symbol` Option)

The `-h` option makes all global symbols static. If you have a symbol that you want to remain global and you use the `-h` option, you can use the `-g` option to declare that symbol to be global. The `-g` option overrides the effect of the `-h` option for the symbol that you specify. The syntax for the `-g` option is:

```
-g= global_symbol
```

### 7.4.9 Make All Global Symbols Static (`-h` Option)

The `-h` option makes all global symbols static. Static symbols are not visible to externally linked modules. By making global symbols static, global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.

The `-h` option effectively nullifies all `.global` assembler directives. All symbols become local to the module in which they are defined, so no external references are possible. For example, assume `file1.obj` and `file2.obj` both define global symbols called `EXT`. By using the `-h` option, you can link these files without conflict. The symbol `EXT` defined in `file1.obj` is treated separately from the symbol `EXT` defined in `file2.obj`.

```
cl6x -z -h file1.obj file2.obj
```

### 7.4.10 Define Heap Size (-heap size Option)

The C/C++ compiler uses an uninitialized section called `.system` for the C run-time memory pool used by `malloc()`. You can set the size of this memory pool at link time by using the `-heap` option. The syntax for the `-heap` option is:

**-heap=** *size*

The *size* must be a constant. This example defines a 4K byte heap:

```
cl6x -z -heap 0x1000 /* defines a 4k heap (.system section)*/
```

The linker creates the `.system` section only if there is a `.system` section in an input file.

The linker also creates a global symbol `__SYSTEM_SIZE` and assigns it a value equal to the size of the heap. The default size is 1K bytes.

For more information, see [Section 7.17, Linking C/C++ Code](#).

### 7.4.11 Alter the Library Search Algorithm (-l Option, -I Option, and C\_DIR/C6X\_C\_DIR Environment Variables)

Usually, when you want to specify a file as linker input, you simply enter the filename; the linker looks for the file in the current directory. For example, suppose the current directory contains the library object.lib. Assume that this library defines symbols that are referenced in the file file1.obj. This is how you link the files:

```
cl6x -z file1.obj object.lib
```

If you want to use a file that is not in the current directory, use the `-l` (lowercase L) linker option. The syntax for this option is:

**-l** [*pathname*] *filename*

The *filename* is the name of an archive, an object file, or linker command file; the space between `-l` and the filename is optional. You can specify up to 128 search paths.

The `-l` option is not required when one or more members of an object library are specified for input to an output section. For more information, see [Section 7.8.7, Allocating an Archive Member to an Output Section](#).

You can augment the linker's directory search algorithm by using the `-l` linker option or the `C_DIR` or `C6X_C_DIR` environment variables. The linker searches for object libraries and command files specified by the `-l` option in the following order:

1. It searches directories named with the `-l` linker option. The `-l` option must appear before the `-l` option on the command line or in a command file.
2. It searches directories named with `C_DIR` and `C6X_C_DIR`.
3. If `C_DIR` and `C6X_C_DIR` are not set, it searches directories named with the assembler's `A_DIR` or `C6X_A_DIR` environment variable.
4. It searches the current directory.

#### 7.4.11.1 Name an Alternate Library Directory (-I pathname Option)

The `-I` option names an alternate directory that contains input files. The syntax for this option is:

**-I** *pathname*

The *pathname* names a directory that contains input files; the space between `-I` and the pathname is optional.

When the linker is searching for input files named with the `-l` option, it searches through directories named with `-I` first. Each `-I` option specifies only one directory, but you can several `-I` options per invocation. When you use the `-I` option to name an alternate directory, it must precede any `-l` option used on the command line or in a command file.

## Linker Options

For example, assume that there are two archive libraries called `r.lib` and `lib2.lib`. The table below shows the directories that `r.lib` and `lib2.lib` reside in, how to set environment variable, and how to use both libraries during a link. Select the row for your operating system:

Operating System	Pathname	Enter
UNIX (Bourne shell)	/ld and /ld2	<code>c16x -z f1.obj f2.obj -I/ld -I/ld2 -lr.lib -llib2.lib</code>
Windows	\ld and \ld2	<code>c16x -z f1.obj f2.obj -I\ld -I\ld2 -lr.lib -llib2.lib</code>

### 7.4.11.2 Name an Alternate Library Directory (C\_DIR and C6X\_C\_DIR Environment Variables)

An environment variable is a system symbol that you define and assign a string to. The linker uses environment variables named `C6X_C_DIR` and `C_DIR` to name alternate directories that contain object libraries. The command syntaxes for assigning the environment variable are:

Operating System	Enter
UNIX (Bourne shell)	<code>C_DIR="pathname<sub>1</sub>; pathname<sub>2</sub>; ..."; export C_DIR</code>
Windows	<code>set C_DIR= pathname<sub>1</sub>; pathname<sub>2</sub>; ...</code>

The *pathnames* are directories that contain input files. Use the `-I` (lowercase L) linker option on the command line or in a command file to tell the linker which library or linker command file to search for. The pathnames must follow these constraints:

- Pathnames must be separated with a semicolon.
- Spaces or tabs at the beginning or end of a path are ignored. For example the space before and after the semicolon in the following is ignored:

```
set C_DIR= c:\path\one\to\tools ; c:\path\two\to\tools
```

- Spaces and tabs are allowed within paths to accommodate Windows directories that contain spaces. For example, the pathnames in the following are valid:

```
set c_DIR=c:\first path\to\tools;d:\second path\to\tools
```

In the example below, assume that two archive libraries called `r.lib` and `lib2.lib` reside in `ld` and `ld2` directories. The table below shows the directories that `r.lib` and `lib2.lib` reside in, how to set the environment variable, and how to use both libraries during a link. Select the row for your operating system:

Operating System	Pathname	Invocation Command
UNIX (Bourne shell)	/ld and /ld2	<code>C_DIR="/ld ;/ld2"; export C_DIRc16x -z f1.obj f2.obj -l r.lib -l lib2.lib</code>
Windows	\ld and \ld2	<code>set C_DIR=\ld;\ld2c16x -z f1.obj f2.obj -l r.lib -l lib2.lib</code>

The environment variable remains set until you reboot the system or reset the variable by entering:

Operating System	Enter
UNIX (Bourne shell)	<code>unset C_DIR</code>
Windows	<code>set C_DIR=</code>

The assembler uses an environment variable named `C6X_A_DIR` or `A_DIR` to name alternate directories that contain copy/include files or macro libraries. If `C6X_C_DIR` or `C_DIR` is not set, the linker searches for object libraries in the directories named with `C6X_A_DIR` or `A_DIR`. For more information about object libraries, see [Section 7.6](#).

### 7.4.12 Disable Conditional Linking (-j Option)

The `-j` option disables removal of unreferenced sections. Only sections marked as candidates for removal with the `.link` assembler directive are affected by conditional linking. See for details on setting up conditional linking using the `.link` directive.



### 7.4.13 Create a Map File (-m filename Option)

The -m option creates a linker map listing and puts it in *filename*. The syntax for the -m option is:

**-m=** *filename*

The linker map describes:

- Memory configuration
- Input and output section allocation
- The addresses of external symbols after they have been relocated

The map file contains the name of the output module and the entry point; it can also contain up to three tables:

- A table showing the new memory configuration if any nondefault memory is specified (memory configuration). The table has the following columns; this information is generated on the basis of the information in the MEMORY directive in the linker command file:
  - **Name.** This is the name of the memory range specified with the MEMORY directive.
  - **Origin.** This specifies the starting address of a memory range.
  - **Length.** This specifies the length of a memory range.
  - **Unused.** This specifies the total amount of unused (available) memory in that memory area.
  - **Attributes.** This specifies one to four attributes associated with the named range:
    - R specifies that the memory can be read.
    - W specifies that the memory can be written to.
    - X specifies that the memory can contain executable code.
    - I specifies that the memory can be initialized.
- For more information about the MEMORY directive, see [Section 7.7, The MEMORY Directive](#).
- A table showing the linked addresses of each output section and the input sections that make up the output sections (section allocation map). This table has the following columns; this information is generated on the basis of the information in the SECTIONS directive in the linker command file:
  - **Output section.** This is the name of the output section specified with the SECTIONS directive.
  - **Origin.** The first origin listed for each output section is the starting address of that output section. The indented origin value is the starting address of that portion of the output section.
  - **Length.** The first length listed for each output section is the length of that output section. The indented length value is the length of that portion of the output section.
  - **Attributes/input sections.** This lists the input file or value associated with an output section. If the input section could not be allocated, the map file will indicate this with "FAILED TO ALLOCATE".
- For more information about the SECTIONS directive, see [Section 7.8, The SECTIONS Directive](#).
- A table showing each external symbol and its address sorted by symbol name.
- A table showing each external symbol and its address sorted by symbol address.

The following example links file1.obj and file2.obj and creates a map file called map.out:

```
cl6x -z file1.obj file2.obj -m=map.out
```

[Example 7-25](#) shows an example of a map file.

### 7.4.14 Name an Output Module (-o Option)

The linker creates an output module when no errors are encountered. If you do not specify a filename for the output module, the linker gives it the default name a.out. If you want to write the output module to a different file, use the -o option. The syntax for the -o option is:

**-o= filename**

The *filename* is the new output module name.

This example links file1.obj and file2.obj and creates an output module named run.out:

```
cl6x -z -o=run.out file1.obj file2.obj
```

#### 7.4.15 Strip Symbolic Information (-s Option)

The -s option creates a smaller output module by omitting symbol table information and line number entries. The -s option is useful for production applications when you do not want to disclose symbolic information to the consumer.

This example links file1.obj and file2.obj and creates an output module, stripped of line numbers and symbol table information, named nosym.out:

```
cl6x -z -o nosym.out -s file1.obj file2.obj
```

Using the -s option limits later use of a symbolic debugger.

#### 7.4.16 Define Stack Size (-stack size Option)

The TMS320C6000 C/C++ compiler uses an uninitialized section, .stack, to allocate space for the run-time stack. You can set the size of this section in bytes at link time with the -stack option. The syntax for the -stack option is:

**-stack size**

The *size* must be a constant and is in bytes. This example defines a 4K byte stack:

```
cl6x -z -stack=0x1000 /* defines a 4K stack (.stack section) */
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size is different.

When the linker defines the .stack section, it also defines a global symbol, \_\_STACK\_SIZE, and assigns it a value equal to the size of the section. The default software stack size is 1K bytes.

#### 7.4.17 Generate Far Call Trampolines (--trampolines Option)

The TMS320C6000 has PC-relative call and PC-relative branch instructions whose range is smaller than the entire address space. When these instructions are used, the destination address must be near enough to the instruction that the difference between the call and the destination fits in the available encoding bits. If the called function is too far away from the calling function, the linker generates an error.

The alternative to a PC-relative call is an absolute call, which is often implemented as an indirect call: load the called address into a register, and call that register. This is often undesirable because it takes more instructions (speed- and size-wise) and requires an extra register to contain the address.

By default, the compiler generates near calls. The --trampolines option causes the linker to generate a trampoline code section for each call that is linked out-of-range of its called destination. The trampoline code section contains a sequence of instructions that performs a transparent long branch to the original called address. Each calling instruction that is out-of-range from the called function is redirected to the trampoline.

For example, in a section of C code the bar function calls the foo function. The compiler generates this code for the function:

```
bar:
    ...
    call    foo      ; call the function "foo"
    ...
```

If the foo function is placed out-of-range from the call to foo that is inside of bar, then with --trampolines the linker changes the original call to foo into a call to foo\_trampoline as shown:

```
bar:
    ...
    call    foo_trampoline ; call a trampoline for foo
    ...
```

The above code generates a trampoline code section called foo\_trampoline, which contains code that executes a long branch to the original called function, foo. For example:

```
foo_trampoline:
    branch_long    foo
```

Trampolines can be shared among calls to the same called function. The only requirement is that all calls to the called function be linked near the called function's trampoline.

When the linker produces a map file (the -m option) and it has produced one or more trampolines, then the map file will contain statistics about what trampolines were generated to reach which functions. A list of calls for each trampoline is also provided in the map file.

---

#### The Linker Assumes B15 Contains the Stack Pointer

**Note:** Assembly language programmers must be aware that the linker assumes B15 contains the stack pointer. The linker must save and restore values on the stack in trampoline code that it generates. If you do not use B15 as the stack pointer, you should use the linker option that disables trampolines, --trampolines=off. Otherwise, trampolines could corrupt memory and overwrite register values.

---

### 7.4.17.1 Carrying Trampolines From Load Space to Run Space

It is sometimes useful to load code in one location in memory and run it in another. The linker provides the capability to specify separate load and run allocations for a section. The burden of actually copying the code from the load space to the run space is left to you.

A copy function must be executed before the real function can be executed in its run space. To facilitate this copy function, the assembler provides the .label directive, which allows you to define a load-time address. These load-time addresses can then be used to determine the start address and size of the code to be copied. However, this mechanism will *not* work if the code contains a call that requires a trampoline to reach its called function. This is because the trampoline code is generated at link time, after the load-time addresses associated with the .label directive have been defined. If the linker detects the definition of a .label symbol in an input section that contains a trampoline call, then a warning is generated.

To solve this problem, you can use the START(), END(), and SIZE() operators (see [Section 7.13.7, Address and Dimension Operators](#)). These operators allow you to define symbols to represent the load-time start address and size inside the linker command file. These symbols can be referenced by the copy code, and their values are not resolved until link time, after the trampoline sections have been allocated.

Here is an example of how you could use the START() and SIZE() operators in association with an output section to copy the trampoline code section along with the code containing the calls that need trampolines:

```
SECTIONS
{
    .foo : load = ROM, run = RAM, start(foo_start), size(foo_size)
        { x.obj(.text) }

    .text: {} > ROM

    .far : { -lrts.lib(.text) } > FAR_MEM
}
```

## Linker Options

---

A function in x.obj contains an run-time-support call. The run-time-support library is placed in far memory and so the call is out-of-range. A trampoline section will be added to the .foo output section by the linker. The copy code can refer to the symbols foo\_start and foo\_size as parameters for the load start address and size of the entire .foo output section. This allows the copy code to copy the trampoline section along with the original x.obj code in .text from its load space to its run space.

### 7.4.17.2 Disadvantages of Using Trampolines

An alternative method to creating a trampoline code section for a call that cannot reach its called function is to actually modify the source code for the call. In some cases this can be done without affecting the size of the code. However, in general, this approach is extremely difficult, especially when the size of the code is affected by the transformation.

While generating far call trampolines provides a more straightforward solution, trampolines have the disadvantage that they are somewhat slower than directly calling a function. They require both a call and a branch. Additionally, while inline code could be tailored to the environment of the call, trampolines are generated in a more general manner, and may be slightly less efficient than inline code.

### 7.4.18 Introduce an Unresolved Symbol (-u symbol Option)

The -u option introduces an unresolved symbol into the linker's symbol table. This forces the linker to search a library and include the member that defines the symbol. The linker must encounter the -u option *before* it links in the member that defines the symbol. The syntax for the -u option is:

**-u=** *symbol*

For example, suppose a library named rts6200.lib contains a member that defines the symbol symtab; none of the object files being linked reference symtab. However, suppose you plan to relink the output module and you want to include the library member that defines symtab in this link. Using the -u option as shown below forces the linker to search rts6200.lib for the member that defines symtab and to link in the member.

```
cl6x -z -u=symtab file1.obj file2.obj rts6200.lib
```

If you do not use -u, this member is not included, because there is no explicit reference to it in file1.obj or file2.obj.

### 7.4.19 Display a Message When an Undefined Output Section Is Created (-w Option)

In a linker command file, you can set up a SECTIONS directive that describes how input sections are combined into output sections. However, if the linker encounters one or more input sections that do not have a corresponding output section defined in the SECTIONS directive, the linker combines the input sections that have the same name into an output section with that name. By default, the linker does not display a message to tell you that this occurred.

You can use the -w option to cause the linker to display a message when it creates a new output section.

For more information about the SECTIONS directive, see [Section 7.8](#). For more information about the default actions of the linker, see [Section 7.12](#).

### 7.4.20 Exhaustively Read and Search Libraries (-x and -priority Options)

There are two ways to exhaustively search for unresolved symbols:

- Reread libraries if you cannot resolve a symbol reference (-x).
- Search libraries in the order that they are specified (-priority).

The linker normally reads input files, including archive libraries, only once when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library, the reference is not resolved.

With the -x option, you can force the linker to reread all libraries. The linker rereads libraries until no more references can be resolved. Linking using -x may be slower, so you should use it only as needed. For example, if a.lib contains a reference to a symbol defined in b.lib, and b.lib contains a reference to a symbol defined in a.lib, you can resolve the mutual dependencies by listing one of the libraries twice, as in:

```
cl6x -z -la.lib -lb.lib -la.lib
```

or you can force the linker to do it for you:

```
cl6x -z -x -la.lib -lb.lib
```

The -priority option provides an alternate search mechanism for libraries. Using -priority causes each unresolved reference to be satisfied by the first library that contains a definition for that symbol. For example:

```
objfile  references A
lib1     defines B
lib2     defines A, B; obj defining A references B

% cl6x -z objfile lib1 lib2
```

Under the existing model, objfile resolves its reference to A in lib2, pulling in a reference to B, which resolves to the B in lib2.

Under -priority, objfile resolves its reference to A in lib2, pulling in a reference to B, but now B is resolved by searching the libraries in order and resolves B to the first definition it finds, namely the one in lib1.

The -priority option is useful for libraries that provide overriding definitions for related sets of functions in other libraries without having to provide a complete version of the whole library.

For example, suppose you want to override versions of malloc and free defined in the rts6200.lib without providing a full replacement for rts6200.lib. Using -priority and linking your new library before rts6200.lib guarantees that all references to malloc and free resolve to the new library.

The -priority option is intended to support linking programs with DSP/BIOS where situations like the one illustrated above occur.

### 7.4.21 Generate XML Link Information File (--xml\_link\_info Option)

The linker supports the generation of an XML link information file via the --xml\_link\_info *file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker generated map file.

See [Appendix C](#), *XML Link Information File Description*, for specifics on the contents of the generated file.

## 7.5 Linker Command Files

Linker command files allow you to put linking information in a file; this is useful when you invoke the linker often with the same information. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line.

Linker command files are ASCII files that contain one or more of the following:

- Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)
- Linker options, which can be used in the command file in the same manner that they are used on the command line
- The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration (see [Section 7.7](#)). The SECTIONS directive controls how sections are built and allocated (see [Section 7.8](#).)
- Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the cl6x -z command and follow it with the name of the command file:

```
cl6x -z command_filename
```

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links the file. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it. Command filenames are case sensitive, regardless of the system used.

[Example 7-1](#) shows a sample linker command file called link.cmd.

### Example 7-1. Linker Command File

```
a.obj          /* First input filename      */
b.obj          /* Second input filename       */
-o prog.out    /* Option to specify output file */
-m prog.map    /* Option to specify map file    */
```

The sample file in [Example 7-1](#) contains only filenames and options. (You can place comments in a command file by delimiting them with /\* and \*/.) To invoke the linker with this command file, enter:

```
cl6x -z link.cmd
```

You can place other parameters on the command line when you use a command file:

```
cl6x -z -r link.cmd c.obj d.obj
```

The linker processes the command file as soon as it encounters the filename, so a.obj and b.obj are linked into the output module before c.obj and d.obj.

You can specify multiple command files. If, for example, you have a file called names.lst that contains filenames and another file called dir.cmd that contains linker directives, you could enter:

```
cl6x -z names.lst dir.cmd
```

One command file can call another command file; this type of nesting is limited to 16 levels. If a command file calls another command file as input, this statement must be the *last* statement in the calling command file.

Blanks and blank lines are insignificant in a command file except as delimiters. This also applies to the format of linker directives in a command file. [Example 7-2](#) shows a sample command file that contains linker directives.

### Example 7-2. Command File With Linker Directives

```
a.obj b.obj c.obj          /* Input filenames      */
-o prog.out -m prog.map    /* Options        */

MEMORY                    /* MEMORY directive */
{
    FAST_MEM:  origin = 0x0100    length = 0x0100
    SLOW_MEM:  origin = 0x7000    length = 0x1000
}

SECTIONS                  /* SECTIONS directive */
{
    .text:    > SLOW_MEM
    .data:    > SLOW_MEM
    .bss:     > FAST_MEM
}
```

For more information, see [Section 7.7](#), *The MEMORY Directive*, and [Section 7.8](#), *The SECTIONS Directive*.

## 7.5.1 Reserved Names in Linker Command Files

The following names are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

align	DSECT	len	o	RUN
ALIGN	f	length	org	SECTIONS
attr	fill	LENGTH	origin	spare
ATTR	FILL	load	ORIGIN	type
block	group	LOAD	range	TYPE
BLOCK	GROUP	MEMORY	run	UNION
COPY	I (lowercase L)	NOLOAD		

## 7.5.2 Constants in Linker Command Files

You can specify constants with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants used in the assembler (see [Section 3.6](#), *Constants*) or the scheme used for integer constants in C syntax.

Examples:

Format	Decimal	Octal	Hexadecimal
Assembler format	32	40q	020h
C format	32	040	0x20



## 7.6 Object Libraries

An object library is a partitioned archive file that contains object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as linker input, the linker includes any members of the library that define existing unresolved symbol references. You can use the archiver to build and maintain libraries. [Chapter 6, Archiver Description](#), contains more information about the archiver.

Using object libraries can reduce link time and the size of the executable module. Normally, if an object file that contains a function is specified at link time, the file is linked whether the function is used or not; however, if that same function is placed in an archive library, the file is included only if the function is referenced.

The order in which libraries are specified is important, because the linker includes only those members that resolve symbols that are undefined at the time the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. Alternatively, you can use the `-x` option to reread libraries until no more references can be resolved (see [Section 7.4.20, Exhaustively Read and Search Libraries \(-x and -priority Options\)](#)). A library has a table that lists all external symbols defined in the library; the linker searches through the table until it determines that it cannot use the library to resolve any more references.

The following examples link several files and libraries, using these assumptions:

- Input files `f1.obj` and `f2.obj` both reference an external function named `clrscr`.
- Input file `f1.obj` references the symbol `origin`.
- Input file `f2.obj` references the symbol `fillclr`.
- Member 0 of library `libc.lib` contains a definition of `origin`.
- Member 3 of library `liba.lib` contains a definition of `fillclr`.
- Member 1 of both libraries defines `clrscr`.

If you enter:

```
cl6x -z f1.obj f2.obj liba.lib libc.lib
```

then:

- Member 1 of `liba.lib` satisfies the `f1.obj` and `f2.obj` references to `clrscr` because the library is searched and the definition of `clrscr` is found.
- Member 0 of `libc.lib` satisfies the reference to `origin`.
- Member 3 of `liba.lib` satisfies the reference to `fillclr`.

If, however, you enter:

```
cl6x -z f1.obj f2.obj libc.lib liba.lib
```

then the references to `clrscr` are satisfied by member 1 of `libc.lib`.

If none of the linked files reference symbols defined in a library, you can use the `-u` option to force the linker to include a library member. (See [Section 7.4.18, Introduce an Unresolved Symbol \(-u symbol Option\)](#).) The next example creates an undefined symbol `route1` in the linker's global symbol table:

```
cl6x -z -u route1 libc.lib
```

If any member of `libc.lib` defines `route1`, the linker includes that member.

Library members are allocated according to the `SECTIONS` directive default allocation algorithm. For more information, see [Section 7.8, The SECTIONS Directive](#).

[Section 7.4.11, Alter the Library Search Algorithm \(-I Option, -I Option, and C\\_DIR/C6X\\_C\\_DIR Environment Variables\)](#) describes methods for specifying directories that contain object libraries.



## 7.7 The MEMORY Directive

The linker determines where output sections are allocated into memory; it must have a model of target memory to accomplish this. The MEMORY directive allows you to specify a model of target memory so that you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections and uses it to determine which memory locations can be used for object code.

The memory configurations of TMS320C6000 systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use MEMORY to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

For more information, see [Section 2.3, How the Linker Handles Sections](#) and [Section 2.4, Relocation](#).

### 7.7.1 Default Memory Model

If you do not use the MEMORY directive, the linker uses a default memory model that is based on the TMS320C6000 architecture. This model assumes that the full 32-bit address space ( $2^{32}$  locations) is present in the system and available for use. For more information about the default memory model, see [Section 7.12, Default Allocation Algorithm](#).

### 7.7.2 MEMORY Directive Syntax

The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each range has several characteristics:

- Name
- Starting address
- Length
- Optional set of attributes
- Optional fill specification

When you use the MEMORY directive, be sure to identify all memory ranges that are available for loading code. Memory defined by the MEMORY directive is configured; any memory that you do not explicitly account for with MEMORY is unconfigured. The linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in [Example 7-3](#) defines a system that has 4K bytes of fast external memory at address 0x0000 0000, 2K bytes of slow external memory at address 0x0000 1000 and 4K bytes of slow external memory at address 0x1000 0000.

#### Example 7-3. The MEMORY Directive

```

/*****
/*      Sample command file with MEMORY directive      */
/*****
file1.obj   file2.obj           /*      Input files      */
-o prog.out                /*      Options        */

MEMORY
{
    FAST_MEM (RX): origin = 0x00000000   length = 0x00001000
    SLOW_MEM (RW): origin = 0x00001000   length = 0x00000800
    EXT_MEM  (RX): origin = 0x10000000   length = 0x00001000

```

The general syntax for the MEMORY directive is:

**MEMORY**

```
{
    name 1 [(attr)] : origin = constant, length = constant [, fill = constant]
    .
    .
    name n [(attr)] : origin = constant, length = constant [, fill = constant]
}
```

<b>name</b>	names a memory range. A memory name can be one to 64 characters; valid characters include A-Z, a-z, \$, ., and _. The names have no special significance to the linker; they simply identify memory ranges. Memory range names are internal to the linker and are not retained in the output file or in the symbol table. All memory ranges must have unique names and must not overlap.
<b>attr</b>	specifies one to four attributes associated with the named range. Attributes are optional; when used, they must be enclosed in parentheses. Attributes restrict the allocation of output sections into certain memory ranges. If you do not use any attributes, you can allocate any output section into any range with no restrictions. Any memory for which no attributes are specified (including all memory in the default model) has all four attributes. Valid attributes are: <ul style="list-style-type: none"> <li><b>R</b> specifies that the memory can be read.</li> <li><b>W</b> specifies that the memory can be written to.</li> <li><b>X</b> specifies that the memory can contain executable code.</li> <li><b>I</b> specifies that the memory can be initialized.</li> </ul>
<b>origin</b>	specifies the starting address of a memory range; enter as <i>origin</i> , <i>org</i> , or <i>o</i> . The value, specified in bytes, is a 32-bit constant and can be decimal, octal, or hexadecimal. specifies the starting address of a memory range; enter as <i>,</i> , or <i>.</i> The value, specified in bytes, is a 32-bit constant and can be decimal, octal, or hexadecimal.
<b>length</b>	specifies the length of a memory range; enter as <i>length</i> , <i>len</i> , or <i>l</i> . The value, specified in bytes, is a 32-bit constant and can be decimal, octal, or hexadecimal. specifies the length of a memory range; enter as <i>,</i> , or <i>.</i> The value, specified in bytes, is a 32-bit constant and can be decimal, octal, or hexadecimal.
<b>fill</b>	specifies a fill character for the memory range; enter as <i>fill</i> or <i>f</i> . Fills are optional. The value is a 32-bit integer constant and can be decimal, octal, or hexadecimal. The fill value is used to fill areas of the memory range that are not allocated to a section. specifies a fill character for the memory range; enter as <i>or</i> . Fills are optional. The value is a 32-bit integer constant and can be decimal, octal, or hexadecimal. The fill value is used to fill areas of the memory range that are not allocated to a section.

---

**Filling Memory Ranges**

**Note:** If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

---

The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFFFFFh:

```
MEMORY
```

```
{
    RFILE (RW) : o = 0x0020h, l = 0x1000, f = 0xFFFFFFFFh
}
```

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control allocation of output sections. After you use MEMORY to specify the target system's memory model, you can use SECTIONS to allocate output sections into specific named memory ranges or into memory that has specific attributes. For example, you could allocate the .text and .data sections into the area named FAST\_MEM and allocate the .bss section into the area named SLOW\_MEM.

## 7.8 The SECTIONS Directive

The SECTIONS directive:

- Describes how input sections are combined into output sections
- Defines output sections in the executable program
- Specifies where output sections are placed in memory (in relation to each other and to the entire memory space)
- Permits renaming of output sections

For more information, see [Section 2.3, How the Linker Handles Sections](#), [Section 2.4, Relocation](#), and [Section 2.2.4, Subsections](#). Subsections allow you to manipulate sections with greater precision.

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. [Section 7.12, Default Allocation Algorithm](#), describes this algorithm in detail.

### 7.8.1 SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

```
SECTIONS
{
    name : [property [, property] [, property] . . . ]
    name : [property [, property] [, property] . . . ]
    name : [property [, property] [, property] . . . ]
}
```

Each section specification, beginning with *name*, defines an output section. (An output section is a section in the output file.) A section name can be a subsection specification. (See [Section 7.8.4](#) for information on multi-level subsections.) After the section name is a list of properties that define the section's contents and how the section is allocated. The properties can be separated by optional commas. Possible properties for a section are as follows:

- **Load allocation** defines where in memory the section is to be loaded.

```
Syntax:    load = allocation      or
           allocation            or
           > allocation
```

- **Run allocation** defines where in memory the section is to be run.

Syntax:     **run = allocation**             or  
              **run > allocation**

- **Input sections** defines the input sections (object files) that constitute the output section.

Syntax:     **{ input\_sections }**

- **Section type** defines flags for special section types.

Syntax:     **type = COPY**                 or  
              **type = DSECT**             or  
              **type = NOLOAD**

For more information, see [Section 7.11](#), *Special Section Types (DSECT, COPY, and NOLOAD)*.

- **Fill value** defines the value used to fill uninitialized holes.

Syntax:     **fill = value**                 or  
              **name :**  
              **[properties=value]**

For more information, see [Section 7.14](#), *Creating and Filling Holes*.

[Example 7-4](#) shows a *SECTIONS* directive in a sample linker command file.

#### **Example 7-4. The *SECTIONS* Directive**

```

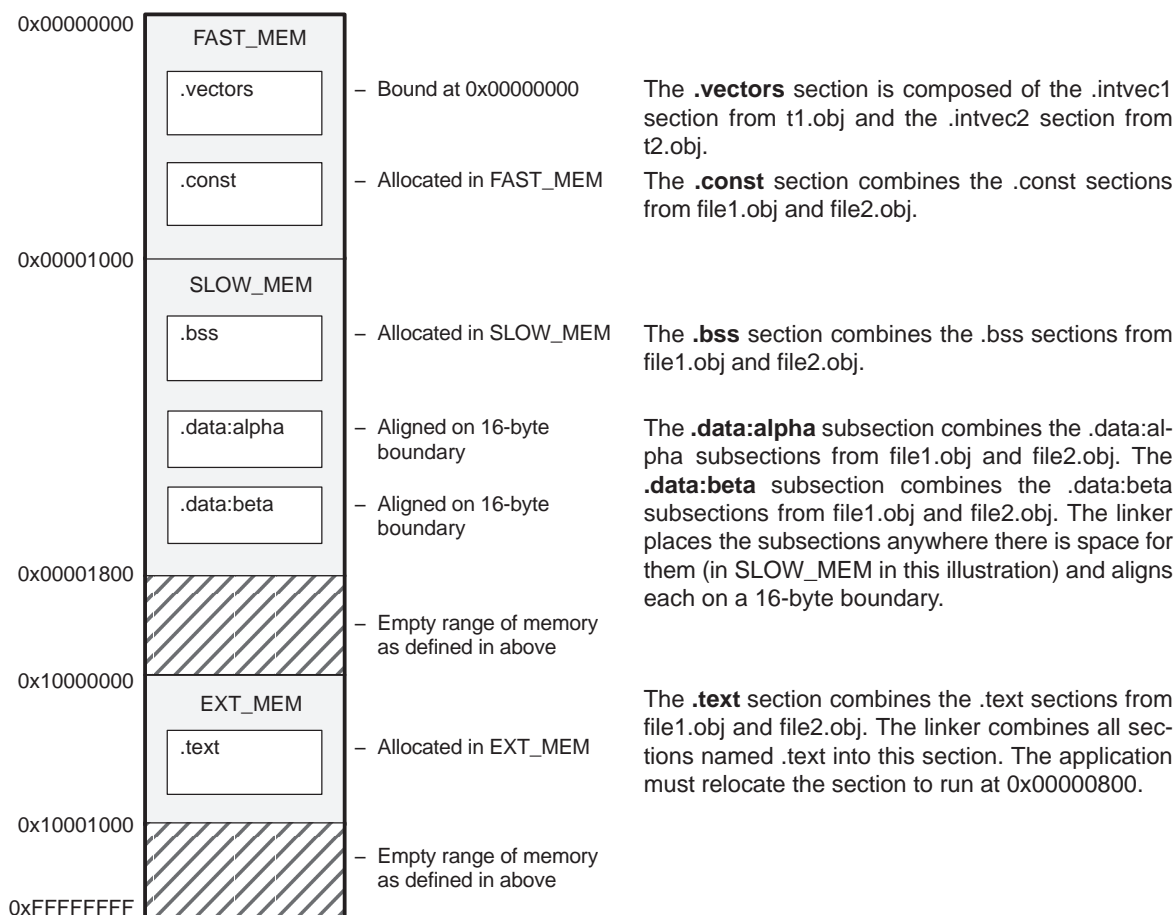
/*****
/* Sample command file with SECTIONS directive */
*****/
file1.obj  file2.obj          /* Input files */
-o prog.out                /* Options */

SECTIONS
{
    .text:      load = EXT_MEM, run = 0x00000800
    .const:     load = FAST_MEM
    .bss:       load = SLOW_MEM
    .vectors:   load = 0x00000000
    {
        t1.obj(.intvec1)
        t2.obj(.intvec2)
        endvec = .;
    }
    .data:alpha: align = 16
    .data:beta:  align = 16
}

```

[Figure 7-2](#) shows the six output sections defined by the *SECTIONS* directive in [Example 7-4](#) (.vectors, .text, .const, .bss, .data:alpha, and .data:beta) and shows how these sections are allocated in memory.

Figure 7-2. Section Allocation Defined by Example 7-4



## 7.8.2 Allocation

The linker assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. The process of locating the output section in the target's memory and assigning its address(es) is called allocation. For more information about using separate load and run allocation, see [Section 7.9, Specifying a Section's Run-Time Address](#).

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to allocate the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run allocation are separate, all parameters following the keyword LOAD apply to load allocation, and those following the keyword RUN apply to run allocation. The allocation parameters are:

**Binding** allocates a section at a specific address.

```
.text: load = 0x1000
```

**Named memory** allocates the section into a range defined in the MEMORY directive with the specified name (like SLOW\_MEM) or attributes.

```
.text: load > SLOW_MEM
```

---

The *SECTIONS Directive*


---

<b>Alignment</b>	uses the align or palign keyword to specify that the section must start on an address boundary. <code>.text: align = 0x100</code>
<b>Blocking</b>	uses the block keyword to specify that the section must fit between two address boundaries: if the section is too big, it starts on an address boundary. <code>.text: block(0x100)</code>

For the load (usually the only) allocation, you can simply use a greater-than sign and omit the load keyword:

```
text: > SLOW_MEM          .text: {...} > SLOW_MEM
.text: > 0x4000
```

If more than one parameter is used, you can string them together as follows:

```
.text: > SLOW_MEM align 16
```

Or if you prefer, use parentheses for readability:

```
.text: load = (SLOW_MEM align(16))
```

You can also use an input section specification to identify the sections from input files that are combined to form an output section. For more information, see [Section 7.8.3, Specifying Input Sections](#).

### 7.8.2.1 Binding

You can supply a specific starting address for an output section by following the section name with an address:

```
.text: 0x00001000
```

This example specifies that the .text section must begin at location 0x1000. The binding address must be a 32-bit constant.

Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

---

#### Binding is Incompatible With Alignment and Named Memory

---

**Note:** You cannot bind a section to an address if you use alignment or named memory. If you try to do this, the linker issues an error message.

---

### 7.8.2.2 Named Memory

You can allocate a section into a memory range that is defined by the MEMORY directive (see [Section 7.7, The MEMORY Directive](#)). This example names ranges and links sections into them:

```
MEMORY
{
    SLOW_MEM (RIX) : origin = 0x00000000, length = 0x00001000
    FAST_MEM (RWIX) : origin = 0x30000000, length = 0x00000300
}

SECTIONS
{
    .text : > SLOW_MEM
    .data : > FAST_MEM ALIGN(128)
    .bss : > FAST_MEM
```

In this example, the linker places .text into the area called SLOW\_MEM. The .data and .bss output sections are allocated into FAST\_MEM. You can align a section within a named memory range; the .data section is aligned on a 128-byte boundary within the FAST\_MEM range.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
    .text: > (X) /* .text --> executable memory */
    .data: > (RI) /* .data --> read or init memory */
    .bss : > (RW) /* .bss --> read or write memory */
}
```

In this example, the .text output section can be linked into either the SLOW\_MEM or FAST\_MEM area because both areas have the X attribute. The .data section can also go into either SLOW\_MEM or FAST\_MEM because both areas have the R and I attributes. The .bss output section, however, must go into the FAST\_MEM area because only FAST\_MEM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no conflicting assignments exist, the .text section starts at address 0. If a section must start on a specific address, use binding instead of named memory.

### 7.8.2.3 Controlling Allocation Using The HIGH Location Specifier

The linker allocates output sections from low to high addresses within a designated memory range by default. Alternatively, you can cause the linker to allocate a section from high to low addresses within a memory range by using the HIGH location specifier in the SECTION directive declaration.

For example, given this MEMORY directive:

```
MEMORY
{
    RAM          : origin = 0x0200, length = 0x0800
    FLASH        : origin = 0x1100, length = 0xEE0
    VECTORS@)    : origin = 0xFFE0, length = 0x001E
    RESET        : origin = 0xFFFE, length = 0x0002
}
```

and an accompanying SECTIONS directive:

```
SECTIONS
{
    .bss      : {} > RAM
    .sysmem   : {} > RAM
    .stack    : {} > RAM (HIGH)
}
```

The HIGH specifier used on the .stack section allocation causes the linker to attempt to allocate .stack into the higher addresses within the RAM memory range. The .bss and .sysmem sections are allocated into the lower addresses within RAM. [Example 7-5](#) illustrates a portion of a map file that shows where the given sections are allocated within RAM for a typical program.

**Example 7-5. Linker Allocation With the HIGH Specifier**

.bss	0	00000200	00000270	UNINITIALIZED
		00000200	0000011a	rts6200.lib : defs.obj (.bss)
		0000031a	00000088	: trgdrv.obj (.bss)
		000003a2	00000078	: lowlev.obj (.bss)
		0000041a	00000046	: exit.obj (.bss)
		00000460	00000008	: memory.obj (.bss)
		00000468	00000004	: _lock.obj (.bss)
		0000046c	00000002	: fopen.obj (.bss)
		0000046e	00000002	hello.obj (.bss)
.system	0	00000470	00000120	UNINITIALIZED
		00000470	00000004	rts6200.lib : memory.obj (.system)
.stack	0	000008c0	00000140	UNINITIALIZED
		000008c0	00000002	rts6200.lib : boot.obj (.stack)

As shown in [Example 7-5](#), the .bss and .system sections are allocated at the lower addresses of RAM (0x0200 - 0x0590) and the .stack section is allocated at address 0x0880, even though lower addresses are available.

Without using the HIGH specifier, the linker allocation would have resulted in the code shown in [Example 7-6](#):

**Example 7-6. Linker Allocation Without HIGH Specifier**

.bss	0	00000200	00000270	UNINITIALIZED
		00000200	0000011a	rts6200.lib : defs.obj (.bss)
		0000031a	00000088	: trgdrv.obj (.bss)
		000003a2	00000078	: lowlev.obj (.bss)
		0000041a	00000046	: exit.obj (.bss)
		00000460	00000008	: memory.obj (.bss)
		00000468	00000004	: _lock.obj (.bss)
		0000046c	00000002	: fopen.obj (.bss)
		0000046e	00000002	hello.obj (.bss)
.stack	0	00000470	00000140	UNINITIALIZED
		00000470	00000002	rts6200.lib : boot.obj (.stack)
.system	0	000005b0	00000120	UNINITIALIZED
		000005b0	00000004	rts6200.lib : memory.obj (.system)

The HIGH specifier is ignored if it is used with specific address binding or automatic section splitting (>> operator).

**7.8.2.4 Alignment and Blocking**

You can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the align keyword. For example:

```
.text: load = align(32)
```

allocates .text so that it falls on a 32-byte boundary.

You can specify the same alignment with the palign keyword. In addition, palign ensures the section's size is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

Blocking is a weaker form of alignment that allocates a section anywhere *within* a block of size n. The specified block size must be a power of 2. For example:



```
bss: load = block(0x0080)
```

allocates .bss so that the entire section is contained in a single 128-byte page or begins on that boundary.

You can use alignment or blocking alone or in conjunction with a memory area, but alignment and blocking cannot be used together.

### 7.8.3 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. In general, the linker combines input sections by concatenating them in the order in which they are specified. However, if alignment or blocking is specified for an input section, all of the input sections within the output section are ordered as follows:

- All aligned sections, from largest to smallest
- All blocked sections, from largest to smallest
- All other sections, from largest to smallest

The size of an output section is the sum of the sizes of the input sections that it comprises.

[Example 7-7](#) shows the most common type of section specification; note that no input sections are listed.

#### Example 7-7. The Most Common Method of Specifying Section Contents

```
SECTIONS
{
    .text:
    .data:
    .bss:
}
```

In [Example 7-7](#), the linker takes all the .text sections from the input files and combines them into the .text output section. The linker concatenates the .text input sections in the order that it encounters them in the input files. The linker performs similar operations with the .data and .bss sections. You can use this type of specification for any output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
    .text :                /* Build .text output section      */
    {
        f1.obj(.text)      /* Link .text section from f1.obj    */
        f2.obj(sec1)       /* Link sec1 section from f2.obj     */
        f3.obj             /* Link ALL sections from f3.obj     */
        f4.obj(.text,sec2) /* Link .text and sec2 from f4.obj   */
    }
}
```

It is not necessary for input sections to have the same name as each other or as the output section they become part of. If a file is listed with no sections, *all* of its sections are included in the output section. If any additional input sections have the same name as an output section but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the linker found more .text sections in the preceding example and these .text sections *were not* specified anywhere in the SECTIONS directive, the linker would concatenate these extra sections after f4.obj(sec2).

The specifications in [Example 7-7](#) are actually a shorthand method for the following:

```
SECTIONS
{
    .text: { *(.text) }
    .data: { *(.data) }
    .bss: { *(.bss) }
}
```

### The SECTIONS Directive

The specification `*(.text)` means *the unallocated .text sections from all the input files*. This format is useful when:

- You want the output section to contain all input sections that have a specified name, but the output section name is different from the input sections' name.
- You want the linker to allocate the input sections before it processes additional input sections or commands within the braces.

The following example illustrates the two purposes above:

```
SECTIONS
{
    .text : {
        abc.obj(xqt)
        *(.text)
    }
    .data : {
        *(.data)
        fil.obj(table)
    }
}
```

In this example, the `.text` output section contains a named section `xqt` from file `abc.obj`, which is followed by all the `.text` input sections. The `.data` section contains all the `.data` input sections, followed by a named section `table` from the file `fil.obj`. This method includes all the unallocated sections. For example, if one of the `.text` input sections was already included in another output section when the linker encountered `*(.text)`, the linker could not include that first `.text` input section in the second output section.

### 7.8.4 Using Multi-Level Subsections

Originally, subsections were identified with the base section name and a subsection name separated by a colon. For example, `A:B` names a subsection of the base section `A`. In certain places in a linker command file specifying a base name, such as `A`, selects the section `A` as well as any subsections of `A`, such as `A:B` or `A:C`.

This concept has been extended to include multiple levels of subsection naming. The original constraints are still true, but a name such as `A:B` can be used to specify a (sub)section of that name as well as any (multi-level) subsections beginning with that name, such as `A:B:C`, `A:B:OTHER`, etc. All the subsections of `A:B` are also subsections of `A`. `A` and `A:B` are supersections of `A:B:C`. Among a group of supersections of a subsection, the nearest supersection is the supersection with the longest name. Thus, among `{A, A:B}` the nearest supersection of `A:B:C:D` is `A:B`.

With multiple levels of subsections, the constraints are the following:

1. When specifying **input** sections within a file (or library unit) the section name selects an input section of the same name and any subsections of that name.
2. Input sections that are not explicitly allocated are allocated in an existing **output** section of the same name or in the nearest existing supersection of such an output section. An exception to this rule is that during a partial link (specified by the `-r` linker option) a subsection is allocated only to an existing output section of the same name.
3. If no such output section described in 2) is defined, the input section is put in a **newly created output** section with the same name as the base name of the input section.

Consider linking input sections with the following names:

europa:north:norway	europa:central:france	europa:south:spain
europa:north:sweden	europa:central:germany	europa:south:italy
europa:north:finland	europa:central:denmark	europa:south:malta
europa:north:iceland		

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS {
    nordic: {*(europe:north)
             *(europe:central:denmark)} /* the nordic countries */
    central: {*(europe:central)}          /* france, germany */
    therest: {*(europe)}                  /* spain, italy, malta */
}
```

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS {
    islands: {*(europe:south:malta)
              *(europe:north:iceland)} /* malta, iceland */
    europe:north:finland : {}          /* finland */
    europe:north         : {}          /* norway, sweden */
    europe:central       : {}          /* germany, denmark */
    europe:central:france: {}          /* france */

    /* (italy, spain) go into a linker-generated output section "europe" */
}
```

---

### Upward Compatibility of Multi-Level Subsections

**Note:** Existing linker commands that use the existing single-level subsection features and which do not contain section names containing multiple colon characters continue to behave as before. However, if section names in a linker command file or in the input sections supplied to the linker contain multiple colon characters, some change in behavior could be possible. You should carefully consider the impact of the new rules for multiple levels to see if it affects a particular system link.

---

## 7.8.5 Allocation Using Multiple Memory Ranges

The linker allows you to specify an explicit list of memory ranges into which an output section can be allocated. Consider the following example:

```
MEMORY
{
    P_MEM1 : origin = 02000h, length = 01000h
    P_MEM2 : origin = 04000h, length = 01000h
    P_MEM3 : origin = 06000h, length = 01000h
    P_MEM4 : origin = 08000h, length = 01000h
}

SECTIONS
{
    .text : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

The | operator is used to specify the multiple memory ranges. The .text output section is allocated as a whole into the first memory range in which it fits. The memory ranges are accessed in the order specified. In this example, the linker first tries to allocate the section in P\_MEM1. If that attempt fails, the linker tries to place the section into P\_MEM2, and so on. If the output section is not successfully allocated in any of the named memory ranges, the linker issues an error message.

With this type of SECTIONS directive specification, the linker can seamlessly handle an output section that grows beyond the available space of the memory range in which it is originally allocated. Instead of modifying the linker command file, you can let the linker move the section into one of the other areas.

### 7.8.6 Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges

The linker can split output sections among multiple memory ranges to achieve an efficient allocation. Use the >> operator to indicate that an output section can be split, if necessary, into the specified memory ranges. For example:

```
MEMORY
{
    P_MEM1 : origin = 02000h, length = 01000h
    P_MEM2 : origin = 04000h, length = 01000h
    P_MEM3 : origin = 06000h, length = 01000h
    P_MEM4 : origin = 08000h, length = 01000h
}

SECTIONS
{
    .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

In this example, the >> operator indicates that the .text output section can be split among any of the listed memory areas. If the .text section grows beyond the available memory in P\_MEM1, it is split on an input section boundary, and the remainder of the output section is allocated to P\_MEM2 | P\_MEM3 | P\_MEM4.

The | operator is used to specify the list of multiple memory ranges.

You can also use the >> operator to indicate that an output section can be split within a single memory range. This functionality is useful when several output sections must be allocated into the same memory range, but the restrictions of one output section cause the memory range to be partitioned. Consider the following example:

```
MEMORY
{
    RAM : origin = 01000h, length = 08000h
}

SECTIONS
{
    .special: { f1.obj(.text) } = 04000h
    .text: { *(.text) } >> RAM
}
```

The .special output section is allocated near the middle of the RAM memory range. This leaves two unused areas in RAM: from 01000h to 04000h, and from the end of f1.obj(.text) to 08000h. The specification for the .text section allows the linker to split the .text section around the .special section and use the available space in RAM on either side of .special.

The >> operator can also be used to split an output section among all memory ranges that match a specified attribute combination. For example:

```
MEMORY
{
    P_MEM1 (RWX) : origin = 01000h, length = 02000h
    P_MEM2 (RWI) : origin = 04000h, length = 01000h
}

SECTIONS
{
    .text: { *(.text) } >> (RW)
}
```

The linker attempts to allocate all or part of the output section into any memory range whose attributes match the attributes specified in the SECTIONS directive.

This SECTIONS directive has the same effect as:

```
SECTIONS
{
    .text: { *(.text) } >> P_MEM1 | P_MEM2}
```

Certain sections should not be split:

- Certain sections created by the compiler, including
  - The .cinit section, which contains the autoinitialization table for C/C++ programs
  - The .pinit section, which contains the list of global constructors for C++ programs
  - The .bss section, which defines global variables
- An output section with an input section specification that includes an expression to be evaluated. The expression may define a symbol that is used in the program to manage the output section at run time.
- An output section that is a GROUP member. The intent of a GROUP directive is to force contiguous allocation of GROUP member output sections.
- An output section that has a START(), END(), OR SIZE() operator applied to it. These operators provide information about a section's load or run address, and size. Splitting the section may compromise the integrity of the operation.
- The run allocation of a UNION. (Splitting the load allocation of a UNION is allowed.)

If you use the >> operator on any of these sections, the linker issues a warning and ignores the operator.

### 7.8.7 Allocating an Archive Member to an Output Section

The ability to specify an archive member of a library archive for allocation into a specific output section can be specified inside angle brackets after a library name. Any object files separated by commas or spaces from the specified archive file are legal within the angle brackets. The syntax for allocating archived library members specifically inside of a SECTIONS directive is as follows:

**[*-l*]** *library name* <*member1*[,*i*] *member2*[,*i*] ...> [*(input sections)*]

**Example 7-8** specifies that the text sections of boot.obj, exit.obj, and strcpy.obj from the run-time-support library should be placed in section .boot. The remainder of the .text sections from the run-time-support library are to be placed in section .rts. Finally, the remainder of all other .text sections are to be placed in section .text.

#### Example 7-8. Archive Members to Output Sections

```
SECTIONS
{
    boot    >          BOOT1
    {
        -lrtsXX.lib<boot.obj> (.text)
        -lrtsXX.lib>exit.obj strcpy.obj> (.text)
    }

    .rts    >          BOOT2
    {
        -lrtsXX.lib (.text)
    }

    .text   >          RAM
    {
        * (.text)
    }
}
```

The -l option (which normally implies a library path search be made for the named file following the option) listed before each library in [Example 7-8](#) is optional when listing specific archive members inside < >. Using < > implies that you are referring to a library.

To collect a set of the input sections from a library in one place, use the -l option within the SECTIONS directive. For example, the following collects all the .text sections from rts6200.lib into the .rtstest section:

```
SECTIONS
{
```

## Specifying a Section's Run-Time Address

---

```
.rtstest { -lrts6200.lib(.text) } > RAM
}
```

---

### SECTIONS Directive Effect on -priority

**Note:** Specifying a library in a SECTIONS directive causes that library to be entered in the list of libraries that the linker searches to resolve references. If you use the -priority option, the first library specified in the command file will be searched first.

---

## 7.9 Specifying a Section's Run-Time Address

At times, you may want to load code into one area of memory and run it in another. For example, you may have performance-critical code in slow external memory. The code must be loaded into slow external memory, but it would run faster in fast external memory.

The linker provides a simple way to accomplish this. You can use the SECTIONS directive to direct the linker to allocate a section twice: once to set its load address and again to set its run address. For example:

```
.fir: load = SLOW_MEM, run = FAST_MEM
```

Use the *load* keyword for the load address and the *run* keyword for the run address.

See [Section 2.5, Run-Time Relocation](#), for an overview on run-time relocation.

### 7.9.1 Specifying Load and Run Addresses

The load address determines where a loader places the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this *does not* happen automatically when you specify a separate run address.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is allocated as if it were two sections of the same size. This means that both allocations occupy space in the memory map and cannot overlay each other or other sections. (The UNION directive provides a way to overlay sections; see [Section 7.10.1, Overlaying Sections With the UNION Statement](#).)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. Everything related to allocation after the keyword *load* affects the load address until the keyword *run* is seen, after which, everything affects the run address. The load and run allocations are completely independent, so any qualification of one (such as alignment) has no effect on the other. You can also specify run first, then load. Use parentheses to improve readability.

The examples below specify load and run addresses:

```
.data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

(align applies only to load)

```
.data: load = (SLOW_MEM align 32), run = FAST_MEM
```

(identical to previous example)

```
.data: run = FAST_MEM, align 32,
      load = align 16
```

(align 32 in FAST\_MEM for run; align 16 anywhere for load)

### 7.9.2 Uninitialized Sections

Uninitialized sections (such as .bss) are not loaded, so their only significant address is the run address. The linker allocates uninitialized sections only once: if you specify both run and load addresses, the linker warns you and ignores the load address. Otherwise, if you specify only one address, the linker treats it as a run address, regardless of whether you call it load or run. This example specifies load and run addresses for an uninitialized section:

```
.bss: load = 0x1000, run = FAST_MEM
```

A warning is issued, load is ignored, and space is allocated in FAST\_MEM. All of the following examples have the same effect. The .bss section is allocated in FAST\_MEM.

```
.bss: load = FAST_MEM
```

```
.bss: run = FAST_MEM
```

```
.bss: > FAST_MEM
```

### 7.9.3 Referring to the Load Address by Using the .label Directive

Normally, any reference to a symbol in a section refers to its run-time address. However, it may be necessary at run time to refer to a load-time address. Specifically, the code that copies a section from its load address to its run address must have access to the load address. The .label directive defines a special symbol that refers to the section's load address. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address. See [Create a Load-Time Address Label](#) for more information on the .label directive.

[Example 7-9](#) and [Example 7-10](#) show the use of the .label directive to copy a section from its load address in SLOW\_MEM to its run address in FAST\_MEM. [Figure 7-3](#) illustrates the run-time execution of [Example 7-9](#).

#### Example 7-9. Copying Section Assembly Language File

```
.sect ".fir"
.align 4
.label fir_src

fir
; insert code here

.label fir_end

.text
MVKL   fir_src, A4
MVKH   fir_src, A4
MVKL   fir_end, A5
MVKH   fir_end, A5
MVKL   fir, A6
MVKH   fir, A6
SUB     A5, A4, A1

loop:
[!A1]  B      done
LDW     *A4+ +, B3
NOP     4
; branch occurs
STW     B3, *A6+ +
SUB     A1, 4, A1
B       loop
NOP     5
; branch occurs

done:
B       fir
NOP     5
; call occurs
```

**Example 7-10. Linker Command File for [Example 7-9](#)**

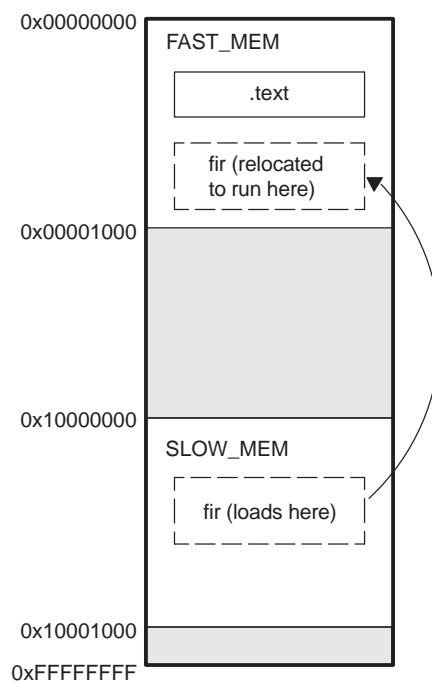
```

/*****
/*      PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE      */
*****/

MEMORY
{
    FAST_MEM :  origin = 0x00001000, length = 0x00001000
    SLOW_MEM :  origin = 0x10000000, length = 0x00001000
}

SECTIONS
{
    .text: load = FAST_MEM
    .fir:  load = SLOW_MEM, run FAST_MEM
}

```

**Figure 7-3. Run-Time Execution of [Example 7-9](#)**



## 7.10 Using UNION and GROUP Statements

Two SECTIONS statements allow you to conserve memory: GROUP and UNION. Unioning sections causes the linker to allocate them to the same run address. Grouping sections causes the linker to allocate them contiguously in memory. Section names can refer to sections, subsections, or archive library members.

### 7.10.1 Overlaying Sections With the UNION Statement

For some applications, you may want to allocate more than one section to run at the same address. For example, you may have several routines you want in fast external memory at various stages of execution. Or you may want several data objects that are not active at the same time to share a block of memory. The UNION statement within the SECTIONS directive provides a way to allocate several sections at the same run-time address.

In [Example 7-11](#), the .bss sections from file1.obj and file2.obj are allocated at the same address in FAST\_MEM. In the memory map, the union occupies as much space as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

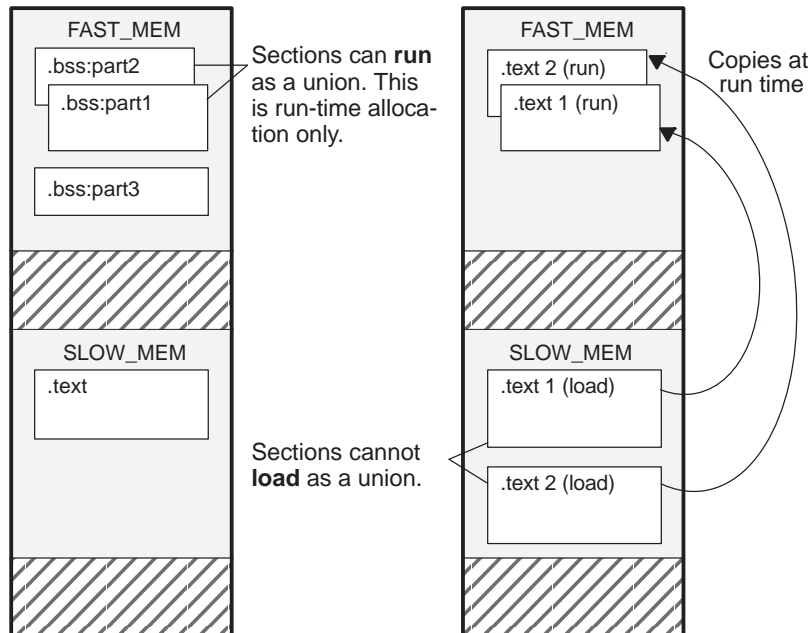
#### Example 7-11. The UNION Statement

```
SECTIONS
{
    .text: load = SLOW_MEM
    UNION: run = FAST_MEM
    {
        .bss:part1: { file1.obj(.bss) }
        .bss:part2: { file2.obj(.bss) }
    }
    .bss:part3: run = FAST_MEM { globals.obj(.bss) }
}
```

Allocation of a section as part of a union affects only its *run address*. Under no circumstances can sections be overlaid for loading. If an initialized section is a union member (an initialized section, such as .text, has raw data), its load allocation *must* be separately specified. See [Example 7-12](#).

#### Example 7-12. Separate Load Addresses for UNION Sections

```
UNION run = FAST_MEM
{
    .text:part1: load = SLOW_MEM, { file1.obj(.text) }
    .text:part2: load = SLOW_MEM, { file2.obj(.text) }
}
```

**Figure 7-4. Memory Allocation Shown in Example 7-11 and Example 7-12**

Since the `.text` sections contain data, they cannot *load* as a union, although they can be *run* as a union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a UNION, the linker issues a warning and allocates load space anywhere it can in configured memory.

Uninitialized sections are not loaded and do not require load addresses.

The UNION statement applies only to allocation of run addresses, so it is meaningless to specify a load address for the union itself. For purposes of allocation, the union is treated as an uninitialized section: any one allocation specified is considered a run address, and if both run and load addresses are specified, the linker issues a warning and ignores the load address.

### 7.10.2 Grouping Output Sections Together

The SECTIONS directive's GROUP option forces several output sections to be allocated contiguously. For example, assume that a section named `term_rec` contains a termination record for a table in the `.data` section. You can force the linker to allocate `.data` and `term_rec` together:

#### Example 7-13. Allocate Sections Together

```
SECTIONS
{
    .text          /* Normal output section          */
    .bss           /* Normal output section          */
    GROUP 0x00001000 : /* Specify a group of sections */
    {
        .data      /* First section in the group      */
        term_rec   /* Allocated immediately after .data */
    }
}
```

You can use binding, alignment, or named memory to allocate a GROUP in the same manner as a single output section. In the preceding example, the GROUP is bound to address `0x00001000`. This means that `.data` is allocated at `0x00001000`, and `term_rec` follows it in memory.

### 7.10.3 Nesting UNIONS and GROUPS

The linker allows arbitrary nesting of GROUP and UNION statements with the SECTIONS directive. By nesting GROUP and UNION statements, you can express hierarchical overlays and groupings of sections. [Example 7-14](#) shows how two overlays can be grouped together.

#### Example 7-14. Nesting GROUP and UNION Statements

```
SECTIONS
{
    GROUP 1000h : run = FAST_MEM
    {
        UNION:
        {
            mysect1:load = SLOW_MEM
            mysect2: load = SLOW_MEM
        }
        UNION:
        {
            mysect3: load = SLOW_MEM
            mysect4: load = SLOW_MEM
        }
    }
}
```

For this example, the linker performs the following allocations:

- The four sections (mysect1, mysect2, mysect3, mysect4) are assigned unique, non-overlapping load addresses in the SLOW\_MEM memory region. This assignment is determined by the particular load allocations given for each section.
- Sections mysect1 and mysect2 are assigned the same run address in FAST\_MEM.
- Sections mysect3 and mysect4 are assigned the same run address in FAST\_MEM.
- The run addresses of mysect1/mysect2 and mysect3/mysect4 are allocated contiguously, as directed by the GROUP statement (subject to alignment and blocking restrictions).

To refer to groups and unions, linker diagnostic messages use the notation:

GROUP\_*n* UNION\_*n*

In this notation, *n* is a sequential number (beginning at 1) that represents the lexical ordering of the group or union in the linker control file, without regard to nesting. Groups and unions each have their own counter.

### 7.10.4 Checking the Consistency of Allocators

The linker checks the consistency of load and run allocations specified for unions, groups, and sections. The following rules are used:

- Run allocations are only allowed for top-level sections, groups, or unions (sections, groups, or unions that are not nested under any other groups or unions). The linker uses the run address of the top-level structure to compute the run addresses of the components within groups and unions.
- The linker does not accept a load allocation for UNIONS.
- The linker does not accept a load allocation for uninitialized sections.
- In most cases, you must provide a load allocation for an initialized section. However, the linker does not accept a load allocation for an initialized section that is located within a group that already defines a load allocator.
- As a shortcut, you can specify a load allocation for an entire group, to determine the load allocations for every initialized section or subgroup nested within the group. However, a load allocation is accepted for an entire group only if all of the following conditions are true:
  - The group is initialized (i.e., it has at least one initialized member).
  - The group is not nested inside another group that has a load allocator.

### Special Section Types (DSECT, COPY, and NOLOAD)

- The group does not contain a union containing initialized sections.
- If the group contains a union with initialized sections, it is necessary to specify the load allocation for each initialized section nested within the group. Consider the following example:

```
SECTIONS
{
    GROUP: load = SLOW_MEM, run = SLOW_MEM
    {
        .text1:
        UNION:
        {
            .text2:
            .text3:
        }
    }
}
```

- The load allocator given for the group does not uniquely specify the load allocation for the elements within the union: .text2 and .text3. In this case, the linker issues a diagnostic message to request that these load allocations be specified explicitly.

## 7.11 Special Section Types (DSECT, COPY, and NOLOAD)

You can assign three special types to output sections: DSECT, COPY, and NOLOAD. These types affect the way that the program is treated when it is linked and loaded. You can assign a type to a section by placing the type after the section definition. For example:

```
SECTIONS
{
    sec1: load = 0x00002000, type = DSECT {f1.obj}
    sec2: load = 0x00004000, type = COPY {f2.obj}
    sec3: load = 0x00006000, type = NOLOAD {f3.obj}
}
```

- The DSECT type creates a dummy section with the following characteristics:
  - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.
  - It can overlay other output sections, other DSECTs, and unconfigured memory.
  - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.
  - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.
  - The section's contents, relocation information, and line number information are not placed in the output module.
- In the preceding example, none of the sections from f1.obj are allocated, but all the symbols are relocated as though the sections were linked at address 0x2000. The other sections can refer to any of the global symbols in sec1.
- A COPY section is similar to a DSECT section, except that its contents and associated information are written to the output module. The .cinit section that contains initialization tables for the TMS320C6000 C/C++ compiler has this attribute under the run-time initialization model.
- A NOLOAD section differs from a normal output section in one respect: the section's contents, relocation information, and line number information are not placed in the output module. The linker allocates space for the section, and it appears in the memory map listing.

## 7.12 Default Allocation Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections that you choose *not* to specify must still be handled by the linker. The linker uses default algorithms to build and allocate sections within the specifications you supply.

If you do not use the MEMORY and SECTIONS directives, the linker allocates output sections as though the definitions in [Example 7-15](#) were specified.

#### Example 7-15. Default Allocation for TMS320C6000 Devices

```
MEMORY
{
    RAM      : origin = 0x00000001, length = 0xFFFFFFFF
}

SECTIONS
{
    .text : ALIGN(32) {} > RAM
    .const : ALIGN(8) {} > RAM
    .data : ALIGN(8) {} > RAM
    .bss : ALIGN(8) {} > RAM
    .cinit : ALIGN(4) {} > RAM ; cflag option only
    .pinit : ALIGN(4) {} > RAM ; cflag option only
    .stack : ALIGN(8) {} > RAM ; cflag option only
    .far : ALIGN(8) {} > RAM ; cflag option only
    .sysmem : ALIGN(8) {} > RAM ; cflag option only
    .switch : ALIGN(4) {} > RAM ; cflag option only
    .cio : ALIGN(4) {} > RAM ; cflag option only
}
```

All .text input sections are concatenated to form a .text output section in the executable output file, and all .data input sections are combined to form a .data output section.

If you use a SECTIONS directive, the linker performs *no part* of the default allocation. Allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described next in [Section 7.12.1](#).

### 7.12.1 How the Allocation Algorithm Creates Output Sections

An output section can be formed in one of two ways:

**Method 1** As the result of a SECTIONS directive definition

**Method 2** By combining input sections with the same name into an output section that is not defined in a SECTIONS directive

If an output section is formed as a result of a SECTIONS directive, this definition completely determines the section's contents. (See [Section 7.8](#), *The SECTIONS Directive*, for examples of how to define an output section's content.)

If an output section is formed by combining input sections not specified by a SECTIONS directive, the linker combines all such input sections that have the same name into an output section with that name. For example, suppose the files f1.obj and f2.obj both contain named sections called Vectors and that the SECTIONS directive does not define an output section for them. The linker combines the two Vectors sections from the input files into a single output section named Vectors, allocates it into memory, and includes it in the output file.

By default, the linker does not display a message when it creates an output section that is not defined in the SECTIONS directive. You can use the -w linker option (see [Section 7.4.19](#), *Display a Message When an Undefined Output Section Is Created (-w Option)*) to cause the linker to display a message when it creates a new output section.

After the linker determines the composition of all output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured. If there is no MEMORY directive, the linker uses the default configuration as shown in [Example 7-15](#). (See [Section 7.7](#), *The MEMORY Directive*, for more information on configuring memory.)

### 7.12.2 Reducing Memory Fragmentation

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. The algorithm comprises these steps:

1. Each output section for which you have supplied a specific binding address is placed in memory at that address.
2. Each output section that is included in a specific, named memory range or that has memory attribute restrictions is allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.
3. Any remaining sections are allocated in the order in which they are defined. Sections not defined in a `SECTIONS` directive are allocated in the order in which they are encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

## 7.13 Assigning Symbols at Link Time

Linker assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to initialize a variable or pointer to an allocation-dependent value.

### 7.13.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the C language:

<i>symbol</i>	=	<i>expression</i> ;	assigns the value of expression to symbol
<i>symbol</i>	+=	<i>expression</i> ;	adds the value of expression to symbol
<i>symbol</i>	-=	<i>expression</i> ;	subtracts the value of expression from symbol
<i>symbol</i>	*=	<i>expression</i> ;	multiplies symbol by expression
<i>symbol</i>	/=	<i>expression</i> ;	divides symbol by expression

The symbol should be defined externally. If it is not, the linker defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in [Section 7.13.3, Assignment Expressions](#). Assignment statements *must* terminate with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Therefore, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol `cur_tab` as the address of the current table. The `cur_tab` symbol must point to either Table1 or Table2. You could accomplish this in the assembly code, but you would need to reassemble the program to change tables. Instead, you can use a linker assignment statement to assign `cur_tab` at link time:

```
prog.obj          /* Input file */
cur_tab = Table1; /* Assign cur_tab to one of the tables */
```

### 7.13.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (`.`), represents the current value of the section program counter (SPC) during allocation. The SPC keeps track of the current location within a section. The linker's `.` symbol is analogous to the assembler's `$` symbol. The `.` symbol can be used only in assignment statements within a `SECTIONS` directive because `.` is meaningful only during allocation and `SECTIONS` controls the allocation process. (See [Section 7.8, The SECTIONS Directive](#).)

The `.` symbol refers to the current run address, not the current load address, of the section.

For example, suppose a program needs to know the address of the beginning of the .data section. By using the .global directive (see [Identify Global Symbols](#)), you can create an external undefined variable called Dstart in the program. Then, assign the value of . to Dstart:

```
SECTIONS
{
    .text:    {}
    .data:    {Dstart = .;}
    .bss :    {}
}
```

This defines Dstart to be the first linked address of the .data section. (Dstart is assigned *before* .data is allocated.) The linker relocates all references to Dstart.

A special type of assignment assigns a value to the . symbol. This adjusts the SPC within an output section and creates a hole between two input sections. Any value assigned to . to create a hole is relative to the beginning of the section, not to the address actually represented by the . symbol. Holes and assignments to . are described in [Section 7.14, Creating and Filling Holes](#).

### 7.13.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in [Table 7-2](#).
- All numbers are treated as long (32-bit) integers.
- Constants are identified by the linker in the same way as by the assembler. That is, numbers are recognized as decimal unless they have a suffix (H or h for hexadecimal and Q or q for octal). C language prefixes are also recognized (0 for octal and 0x for hex). Hexadecimal constants must begin with a digit. No binary constants are allowed.
- Symbols within an expression have only the value of the symbol's *address*. No type-checking is performed.
- Linker expressions can be absolute or relocatable. If an expression contains *any* relocatable symbols (and 0 or more constants or absolute symbols), it is relocatable. Otherwise, the expression is absolute. If a symbol is assigned the value of a relocatable expression, it is relocatable; if it is assigned the value of an absolute expression, it is absolute.

The linker supports the C language operators listed in [Table 7-2](#) in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in [Table 7-2](#), the linker also has an align operator that allows a symbol to be aligned on an n-byte boundary within an output section (n is a power of 2). For example, the expression

```
. = align(16);
```

aligns the SPC within the current section on the next 16-byte boundary. Because the align operator is a function of the current SPC, it can be used only in the same context as . —that is, within a SECTIONS directive.



**Table 7-2. Groups of Operators Used in Expressions (Precedence)**

Group 1 (Highest Precedence)		Group 6	
!	Logical NOT	&	Bitwise AND
~	Bitwise NOT		
-	Negation		
Group 2		Group 7	
*	Multiplication		Bitwise OR
/	Division		
%	Modulus		
Group 3		Group 8	
+	Addition	&&	Logical AND
-	Subtraction		
Group 4		Group 9	
>>	Arithmetic right shift		Logical OR
<<	Arithmetic left shift		
Group 5		Group 10 (Lowest Precedence)	
==	Equal to	=	Assignment
!=	Not equal to	+=	A += B " A = A + B
>	Greater than	-=	A -= B " A = A - B
<	Less than	*=	A *= B " A = A * B
<=	Less than or equal to	/=	A /= B " A = A / B
>=	Greater than or equal to		

### 7.13.4 Symbols Defined by the Linker

The linker automatically defines several symbols based on which sections are used in your assembly source. A program can use these symbols at run time to determine where a section is linked. Since these symbols are external, they appear in the linker map. Each symbol can be accessed in any assembly language module if it is declared with a `.global` directive (see [Identify Global Symbols](#)). You must have used the corresponding section in a source module for the symbol to be created. Values are assigned to these symbols as follows:

- .text** is assigned the first address of the `.text` output section.  
(It marks the *beginning* of executable code.)
- etext** is assigned the first address following the `.text` output section.  
(It marks the *end* of executable code.)
- .data** is assigned the first address of the `.data` output section.  
(It marks the *beginning* of initialized data tables.)
- edata** is assigned the first address following the `.data` output section.  
(It marks the *end* of initialized data tables.)
- .bss** is assigned the first address of the `.bss` output section.  
(It marks the *beginning* of uninitialized data.)
- end** is assigned the first address following the `.bss` output section.  
(It marks the *end* of uninitialized data.)

The following symbols are defined only for C/C++ support when the `-c` or `-cr` option is used.

- \_\_STACK\_SIZE** is assigned the size of the `.stack` section.
- \_\_SYSMEM\_SIZE** is assigned the size of the `.sysmem` section.



### 7.13.5 Assigning Exact Start, End, and Size Values of a Section to a Symbol

The code generation tools currently support the ability to load program code in one area of (slow) memory and run it in another (faster) area. This is done by specifying separate load and run addresses for an output section or group in the linker command file. Then execute a sequence of instructions (the copying code in [Example 7-9](#)) that moves the program code from its load area to its run area before it is needed.

There are several responsibilities that a programmer must take on when setting up a system with this feature. One of these responsibilities is to determine the size and run-time address of the program code to be moved. The current mechanisms to do this involve use of the `.label` directives in the copying code. A simple example is illustrated [Example 7-9](#).

This method of specifying the size and load address of the program code has limitations. While it works fine for an individual input section that is contained entirely within one source file, this method becomes more complicated if the program code is spread over several source files or if the programmer wants to copy an entire output section from load space to run space.

Another problem with this method is that it does not account for the possibility that the section being moved may have an associated far call trampoline section that needs to be moved with it.

### 7.13.6 Why the Dot Operator Does Not Always Work

The dot operator (`.`) is used to define symbols at link-time with a particular address inside of an output section. It is interpreted like a PC. Whatever the current offset within the current section is, that is the value associated with the dot. Consider an output section specification within a `SECTIONS` directive:

```
outsect:
{
    s1.obj(.text)
    end_of_s1 = .;
    start_of_s2 = .;
    s2.obj(.text)
    end_of_s2 = .;
}
```

This statement creates three symbols:

- `end_of_s1`—the end address of `.text` in `s1.obj`
- `start_of_s2`—the start address of `.text` in `s2.obj`
- `end_of_s2`—the end address of `.text` in `s2.obj`

Suppose there is padding between `s1.obj` and `s2.obj` that is created as a result of alignment. Then `start_of_s2` is not really the start address of the `.text` section in `s2.obj`, but it is the address before the padding needed to align the `.text` section in `s2.obj`. This is due to the linker's interpretation of the dot operator as the current PC. It is also due to the fact that the dot operator is evaluated independently of the input sections around it.

Another potential problem in the above example is that `end_of_s2` may not account for any padding that was required at the end of the output section. You cannot reliably use `end_of_s2` as the end address of the output section. One way to get around this problem is to create a dummy section immediately after the output section in question. For example:

```
GROUP
{
    outsect:
    {
        start_of_outsect = .;
        ...
    }
    dummy: { size_of_outsect = . - start_of_outsect; }
```

### 7.13.7 Address and Dimension Operators

Six new operators have been added to the linker command file syntax:

<b>LOAD_START( sym )</b> <b>START( sym )</b>	Defines <i>sym</i> with the load-time start address of related allocation unit
<b>LOAD_END( sym )</b> <b>END( sym )</b>	Defines <i>sym</i> with the load-time end address of related allocation unit
<b>LOAD_SIZE( sym )</b> <b>SIZE( sym )</b>	Defines <i>sym</i> with the load-time size of related allocation unit
<b>RUN_START( sym )</b>	Defines <i>sym</i> with the run-time start address of related allocation unit
<b>RUN_END( sym )</b>	Defines <i>sym</i> with the run-time end address of related allocation unit
<b>RUN_SIZE( sym )</b>	Defines <i>sym</i> with the run-time size of related allocation unit

---

#### Linker Command File Operator Equivalencies

**Note:** LOAD\_START() and START() are equivalent, as are LOAD\_END()/END() and LOAD\_SIZE()/SIZE().

---

The new address and dimension operators can be associated with several different kinds of allocation units, including input items, output sections, GROUPs, and UNIONs. The following sections provide some examples of how the operators can be used in each case.

#### 7.13.7.1 Input Items

Consider an output section specification within a SECTIONS directive:

```
outsect:
{
    s1.obj(.text)
    end_of_s1 = .;
    start_of_s2 = .;
    s2.obj(.text)
    end_of_s2 = .;
}
```

This can be rewritten using the START and END operators as follows:

```
outsect:
{
    s1.obj(.text) { END(end_of_s1) }
    s2.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

The values of end\_of\_s1 and end\_of\_s2 will be the same as if you had used the dot operator in the original example, but start\_of\_s2 would be defined after any necessary padding that needs to be added between the two .text sections. Remember that the dot operator would cause start\_of\_s2 to be defined before any necessary padding is inserted between the two input sections.

The syntax for using these operators in association with input sections calls for braces { } to enclose the operator list. The operators in the list are applied to the input item that occurs immediately before the list.

### 7.13.7.2 Output Section

The START, END, and SIZE operators can also be associated with an output section. Here is an example:

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
    <list of input items>
}
```

In this case, the SIZE operator defines size\_of\_outsect to incorporate any padding that is required in the output section to conform to any alignment requirements that are imposed.

The syntax for specifying the operators with an output section do not require braces to enclose the operator list. The operator list is simply included as part of the allocation specification for an output section.

### 7.13.7.3 GROUPs

Here is another use of the START and SIZE operators in the context of a GROUP specification:

```
GROUP
{
    outsect1: { ... }
    outsect2: { ... }
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

This can be useful if the whole GROUP is to be loaded in one location and run in another. The copying code can use group\_start and group\_size as parameters for where to copy from and how much is to be copied. This makes the use of .label in the source code unnecessary.

### 7.13.7.4 UNIONS

The RUN\_SIZE and LOAD\_SIZE operators provide a mechanism to distinguish between the size of a UNION's load space and the size of the space where its constituents are going to be copied before they are run. Here is an example:

```
UNION: run = RAM, LOAD_START(union_load_addr),
      LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
    .text1: load = ROM, SIZE(text1_size) { f1.obj(.text) }
    .text2: load = ROM, SIZE(text2_size) { f2.obj(.text) }
}
```

Here union\_ld\_sz is going to be equal to the sum of the sizes of all output sections placed in the union. The union\_run\_sz value is equivalent to the largest output section in the union. Both of these symbols incorporate any padding due to blocking or alignment requirements.

## 7.14 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called *holes*. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles holes and how you can fill holes (and uninitialized sections) with values.

### 7.14.1 Initialized and Uninitialized Sections

There are two rules to remember about the contents of output sections. An output section contains either:

- Raw data for the *entire* section
- No raw data

A section that has raw data is referred to as *initialized*. This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The `.text` and `.data` sections *always* have raw data if anything was assembled into them. Named sections defined with the `.sect` assembler directive also have raw data.

By default, the `.bss` section (see [the .bss topic](#)) and sections defined with the `.usect` directive (see [the .usect topic](#)) have no raw data (they are *uninitialized*). They occupy space in the memory map but have no actual contents. Uninitialized sections typically reserve space in fast external memory for variables. In the object file, an uninitialized section has a normal section header and can have symbols defined in it; no memory image, however, is stored in the section.

### 7.14.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must supply raw data for the hole*.

Holes can be created only *within* output sections. Space can exist *between* output sections, but such space is not a hole. To fill the space between output sections, see [Section 7.7.2, MEMORY Directive Syntax](#).

To create a hole in an output section, you must use a special type of linker assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by `.`) by adding to it, assigning a greater value to it, or aligning it on an address boundary. The operators, expressions, and syntaxes of assignment statements are described in [Section 7.13, Assigning Symbols at Link Time](#).

The following example uses assignment statements to create holes in output sections:

```
SECTIONS
{
    outsect:
    {
        file1.obj(.text)
        . += 0x0100 /* Create a hole with size 0x0100 */
        file2.obj(.text)
        . = align(16); /* Create a hole to align the SPC */
        file3.obj(.text)
    }
}
```

The output section `outsect` is built as follows:

1. The `.text` section from `file1.obj` is linked in.
2. The linker creates a 256-byte hole.
3. The `.text` section from `file2.obj` is linked in after the hole.
4. The linker creates another hole by aligning the SPC on a 16-byte boundary.
5. Finally, the `.text` section from `file3.obj` is linked in.

All values assigned to the `.` symbol within a section refer to the *relative address within the section*. The

linker handles assignments to the `.` symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement `. = align(16)` in the example. This statement effectively aligns the `file3.obj .text` section to start on a 16-byte boundary within `outsect`. If `outsect` is ultimately allocated to start on an address that is not aligned, the `file3.obj .text` section will not be aligned either.

The `.` symbol refers to the current run address, not the current load address, of the section.

Expressions that decrement the `.` symbol are illegal. For example, it is invalid to use the `-=` operator in an assignment to the `.` symbol. The most common operators used in assignments to the `.` symbol are `+=` and `align`.

If an output section contains all input sections of a certain type (such as `.text`), you can use the following statements to create a hole at the beginning or end of the output section.

```
.text:    { . += 0x0100; }      /* Hole at the beginning */
.data:    { *(<code>.data)
          . += 0x0100; }      /* Hole at the end          */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* The following example illustrates this method:

```
SECTIONS
{
    outsect:
    {
        file1.obj(.text)
file1.obj(.bss)          /* This becomes a hole */
    }
}
```

Because the `.text` section has raw data, all of `outsect` must also contain raw data. Therefore, the uninitialized `.bss` section becomes a hole.

Uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

### 7.14.3 Filling Holes

When a hole exists in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 32-bit fill value that is replicated through memory until it fills the hole. The linker determines the fill value as follows:

1. If the hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an `=` sign and a 32-bit constant. For example:

```
SECTIONS
{ outsect:
  {
    file1.obj(.text)
    file2.obj(.bss)= 0xFF00FF00 /* Fill this hole with 0xFF00FF00 */
  }
}
```

2. You can also specify a fill value for all the holes in an output section by supplying the fill value after the section definition:

```
SECTIONS
{ outsect:fill = 0xFF00FF00 /* Fills holes with 0xFF00FF00 */
  {
    . += 0x0010;             /* This creates a hole          */
    file1.obj(.text)
    file1.obj(.bss)          /* This creates another hole */
  }
}
```

3. If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with the `-f` option (see [Section 7.4.7](#), *Set Default Fill Value (-f fill\_value Option)*). For example, suppose

## Linker-Generated Copy Tables

---

the command file link.cmd contains the following SECTIONS directive:

```
SECTIONS
{
    .text: { .= 0x0100; } /* Create a 100 word hole */
}
```

Now invoke the linker with the -f option:

```
cl6x -z -f 0xFFFFFFFF link.cmd
```

This fills the hole with 0xFFFFFFFF.

- If you do not invoke the linker with the -f option or otherwise specify a fill value, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

### 7.14.4 Explicit Initialization of Uninitialized Sections

You can force the linker to initialize an uninitialized section by specifying an explicit fill value for it in the SECTIONS directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
    .bss: fill = 0x12341234 /* Fills .bss with 0x12341234 */
}
```

---

#### Filling Sections

**Note:** Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large sections or holes.

---

## 7.15 Linker-Generated Copy Tables

The linker supports extensions to the linker command file syntax that enable the following:

- Make it easier for you to copy objects from load-space to run-space at boot time
- Make it easier for you to manage memory overlays at run time
- Allow you to split GROUPs and output sections that have separate load and run addresses

### 7.15.1 A Current Boot-Loaded Application Development Process

In some embedded applications, there is a need to copy or download code and/or data from one location to another at boot time before the application actually begins its main execution thread. For example, an application may have its code and/or data in FLASH memory and need to copy it into on-chip memory before the application begins execution.

One way you can develop an application like this is to create a copy table in assembly code that contains three elements for each block of code or data that needs to be moved from FLASH into on-chip memory at boot time:

- The load address
- The run address
- The size

The process you follow to develop such an application might look like this:

1. Build the application to produce a .map file that contains the load and run addresses of each section that has a separate load and run placement.
2. Edit the copy table (used by the boot loader) to correct the load and run addresses as well as the size of each block of code or data that needs to be moved at boot time.
3. Build the application again, incorporating the updated copy table.
4. Run the application.

This process puts a heavy burden on you to maintain the copy table (by hand, no less). Each time a piece of code or data is added or removed from the application, you must repeat the process in order to keep the contents of the copy table up to date.

### 7.15.2 An Alternative Approach

You can avoid some of this maintenance burden by using the `LOAD_START()`, `RUN_START()`, and `SIZE()` operators that are already part of the linker command file syntax. For example, instead of building the application to generate a .map file, the linker command file can be annotated:

```
SECTIONS
{
    .flashcode: { app_tasks.obj(.text) }
        load = FLASH, run = PMEM,
        LOAD_START(_flash_code_ld_start),
        RUN_START(_flash_code_rn_start),
        SIZE(_flash_code_size)

    ...
}
```

In this example, the `LOAD_START()`, `RUN_START()`, and `SIZE()` operators instruct the linker to create three symbols:

Symbol	Description
<code>_flash_code_ld_start</code>	Load address of .flashcode section
<code>_flash_code_rn_start</code>	Run address of .flashcode section
<code>_flash_code_size</code>	Size of .flashcode section

These symbols can then be referenced from the copy table. The actual data in the copy table will be updated automatically each time the application is linked. This approach removes step 1 of the process described in [Section 7.15.1](#).

While maintenance of the copy table is reduced markedly, you must still carry the burden of keeping the copy table contents in sync with the symbols that are defined in the linker command file. Ideally, the linker would generate the boot copy table automatically. This would avoid having to build the application twice *and* free you from having to explicitly manage the contents of the boot copy table.

For more information on the `LOAD_START()`, `RUN_START()`, and `SIZE()` operators, see [Section 7.13.7, Address and Dimension Operators](#).

### 7.15.3 Overlay Management Example

Consider an application which contains a memory overlay that must be managed at run time. The memory overlay is defined using a UNION in the linker command file as illustrated in [Example 7-16](#):

#### Example 7-16. Using a UNION for Memory Overlay

```
SECTIONS
{
    ...

    UNION
    {
        GROUP
        {
            .task1: { task1.obj(.text) }
            .task2: { task2.obj(.text) }

        } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)

        GROUP
        {
            .task3: { task3.obj(.text) }
            .task4: { task4.obj(.text) }

        } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)

    } run = RAM, RUN_START(_task_run_start)

    ...
}
```

The application must manage the contents of the memory overlay at run time. That is, whenever any services from .task1 or .task2 are needed, the application must first ensure that .task1 and .task2 are resident in the memory overlay. Similarly for .task3 and .task4.

To affect a copy of .task1 and .task2 from ROM to RAM at run time, the application must first gain access to the load address of the tasks (\_task12\_load\_start), the run address (\_task\_run\_start), and the size (\_task12\_size). Then this information is used to perform the actual code copy.

### 7.15.4 Generating Copy Tables Automatically With the Linker

The linker supports extensions to the linker command file syntax that enable you to do the following:

- Identify any object components that may need to be copied from load space to run space at some point during the run of an application
- Instruct the linker to automatically generate a copy table that contains (at least) the load address, run address, and size of the component that needs to be copied
- Instruct the linker to generate a symbol specified by you that provides the address of a linker-generated copy table. For instance, [Example 7-16](#) can be written as shown in [Example 7-17](#):



### Example 7-17. Produce Address for Linker Generated Copy Table

```
SECTIONS
{
    ...

    UNION
    {
        GROUP
        {
            .task1: { task1.obj(.text) }
            .task2: { task2.obj(.text) }

        } load = ROM, table(_task12_copy_table)

        GROUP
        {
            .task3: { task3.obj(.text) }
            .task4: { task4.obj(.text) }

        } load = ROM, table(_task34_copy_table)

    } run = RAM
    ...
}
```

Using the SECTIONS directive from [Example 7-17](#) in the linker command file, the linker generates two copy tables named: `_task12_copy_table` and `_task34_copy_table`. Each copy table provides the load address, run address, and size of the GROUP that is associated with the copy table. This information is accessible from application source code using the linker-generated symbols, `_task12_copy_table` and `_task34_copy_table`, which provide the addresses of the two copy tables, respectively.

Using this method, you do not have to worry about the creation or maintenance of a copy table. You can reference the address of any copy table generated by the linker in C/C++ or assembly source code, passing that value to a general purpose copy routine which will process the copy table and affect the actual copy.

#### 7.15.5 The `table()` Operator

You can use the `table()` operator to instruct the linker to produce a copy table. A `table()` operator can be applied to an output section, a GROUP, or a UNION member. The copy table generated for a particular `table()` specification can be accessed through a symbol specified by you that is provided as an argument to the `table()` operator. The linker creates a symbol with this name and assigns it the address of the copy table as the value of the symbol. The copy table can then be accessed from the application using the linker-generated symbol.

Each `table()` specification you apply to members of a given UNION must contain a unique name. If a `table()` operator is applied to a GROUP, then none of that GROUP's members may be marked with a `table()` specification. The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the `table()` specification. The linker does not generate a copy table for erroneous `table()` operator specifications.

### 7.15.6 Boot-Time Copy Tables

The linker supports a special copy table name, BINIT (or binit), that you can use to create a boot-time copy table. For example, the linker command file for the boot-loaded application described in [Section 7.15.2](#) can be rewritten as follows:

```
SECTIONS
{
    .flashcode: { app_tasks.obj(.text) }
    load = FLASH, run = PMEM,
        table(BINIT)
    ...
}
```

For this example, the linker creates a copy table that can be accessed through a special linker-generated symbol, `__binit__`, which contains the list of all object components that need to be copied from their load location to their run location at boot-time. If a linker command file does not contain any uses of `table(BINIT)`, then the `__binit__` symbol is given a value of -1 to indicate that a boot-time copy table does not exist for a particular application.

You can apply the `table(BINIT)` specification to an output section, GROUP, or UNION member. If used in the context of a UNION, only one member of the UNION can be designated with `table(BINIT)`. If applied to a GROUP, then none of that GROUP's members may be marked with `table(BINIT)`. The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the `table(BINIT)` specification.

### 7.15.7 Using the table() Operator to Manage Object Components

If you have several pieces of code that need to be managed together, then you can apply the same `table()` operator to several different object components. In addition, if you want to manage a particular object component in multiple ways, you can apply more than one `table()` operator to it. Consider the linker command file excerpt in [Example 7-18](#):

**Example 7-18. Linker Command File to Manage Object Components**

```
SECTIONS
{
    UNION
    {
        .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

        .second: { a2.obj(.text), b2.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
    }

    .extra: load = EMEM, run = PMEM, table(BINIT)
    ...
}
```

In this example, the output sections `.first` and `.extra` are copied from external memory (EMEM) into program memory (PMEM) at boot time while processing the BINIT copy table. After the application has started executing its main thread, it can then manage the contents of the overlay using the two overlay copy tables named: `_first_ctbl` and `_second_ctbl`.

### 7.15.8 Copy Table Contents

In order to use a copy table that is generated by the linker, you must be aware of the contents of the copy table. This information is included in a new run-time-support library header file, `cpy_tbl.h`, which contains a C source representation of the copy table data structure that is automatically generated by the linker.

[Example 7-19](#) shows the TMS320C6000 copy table header file.

**Example 7-19. TMS320C6000 cpy\_tbl.h File**

```

/*****
 * cpy_tbl.h
 *
 * Copyright ©) 2003 Texas Instruments Incorporated
 *
 * Specification of copy table data structures which can be automatically
 * generated by the linker (using the table() operator in the LCF).
 *
 *****/

/*****
 * Copy Record Data Structure
 *****/
typedef struct copy_record
{
    unsigned int load_addr;
    unsigned int run_addr;
    unsigned int size;
} COPY_RECORD;

/*****
 * Copy Table Data Structure
 *****/
typedef struct copy_table
{
    unsigned short rec_size;
    unsigned short num_recs;
    COPY_RECORD   recs[1];
} COPY_TABLE;

/*****
 * Prototype for general purpose copy routine.
 *****/
extern void copy_in(COPY_TABLE *tp);

```

For each object component that is marked for a copy, the linker creates a COPY\_RECORD object for it. Each COPY\_RECORD contains at least the following information for the object component:

- The load address
- The run address
- The size

The linker collects all COPY\_RECORDs that are associated with the same copy table into a COPY\_TABLE object. The COPY\_TABLE object contains the size of a given COPY\_RECORD, the number of COPY\_RECORDs in the table, and the array of COPY\_RECORDs in the table. For instance, in the BINIT example in [Section 7.15.6](#), the .first and .extra output sections will each have their own COPY\_RECORD entries in the BINIT copy table. The BINIT copy table will then look like this:

```

COPY_TABLE __binit__ = { 12, 2,
    { <load address of .first>,
      <run address of .first>,
      <size of .first> },
    { <load address of .extra>,
      <run address of .extra>,
      <size of .extra> } };

```

### 7.15.9 General Purpose Copy Routine

The `cpy_tbl.h` file in [Example 7-19](#) also contains a prototype for a general-purpose copy routine, `copy_in()`, which is provided as part of the run-time-support library. The `copy_in()` routine takes a single argument: the address of a linker-generated copy table. The routine then processes the copy table data object and performs the copy of each object component specified in the copy table.

The `copy_in()` function definition is provided in the `cpy_tbl.c` run-time-support source file shown in [Example 7-20](#).

#### Example 7-20. Run-Time-Support `cpy_tbl.c` File

```

/*****
/* cpy_tbl.c
/*
/* Copyright ©) 2003 Texas Instruments Incorporated
/*
/* General purpose copy routine. Given the address of a linker-generated
/* COPY_TABLE data structure, effect the copy of all object components
/* that are designated for copy via the corresponding LCF table() operator.
/*
*****/
#include <cpy_tbl.h>
#include <string.h>

/*****
/* COPY_IN()
*****/
void copy_in(COPY_TABLE *tp)
{
    unsigned short I;
    for (I = 0; I < tp->num_recs; I++)
    {
        COPY_RECORD crp = tp->recs[i];
        unsigned char *ld_addr = (unsigned char *)crp.load_addr;
        unsigned char *rn_addr = (unsigned char *)crp.run_addr;
        memcpy(rn_addr, ld_addr, crp.size);
    }
}

```

### 7.15.10 Linker Generated Copy Table Sections and Symbols

The linker creates and allocates a separate input section for each copy table that it generates. Each copy table symbol is defined with the address value of the input section that contains the corresponding copy table.

The linker generates a unique name for each overlay copy table input section. For example, `table(_first_ctbl)` would place the copy table for the `.first` section into an input section called `.ovly:_first_ctbl`. The linker creates a single input section, `.binit`, to contain the entire boot-time copy table.

[Example 7-21](#) illustrates how you can control the placement of the linker-generated copy table sections using the input section names in the linker command file.

### Example 7-21. Controlling the Placement of the Linker-Generated Copy Table Sections

```
SECTIONS
{
    UNION
    {
        .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
        load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

        .second: { a2.obj(.text), b2.obj(.text) }
        load = EMEM, run = PMEM, table(_second_ctbl)
    }

    .extra: load = EMEM, run = PMEM, table(BINIT)

    ...

    .ovly: { } > BMEM
    .binit: { } > BMEM
}
```

For the linker command file in [Example 7-21](#), the boot-time copy table is generated into a .binit input section, which is collected into the .binit output section, which is mapped to an address in the BMEM memory area. The \_first\_ctbl is generated into the .ovly:\_first\_ctbl input section and the \_second\_ctbl is generated into the .ovly:\_second\_ctbl input section. Since the base names of these input sections match the name of the .ovly output section, the input sections are collected into the .ovly output section, which is then mapped to an address in the BMEM memory area.

If you do not provide explicit placement instructions for the linker-generated copy table sections, they are allocated according to the linker's default placement algorithm.

The linker does not allow other types of input sections to be combined with a copy table input section in the same output section. The linker does not allow a copy table section that was created from a partial link session to be used as input to a succeeding link session.

#### 7.15.11 Splitting Object Components and Overlay Management

In previous versions of the linker, splitting sections that have separate load and run placement instructions was not permitted. This restriction was because there was no effective mechanism for you, the developer, to gain access to the load address or run address of each one of the pieces of the split object component. Therefore, there was no effective way to write a copy routine that could move the split section from its load location to its run location.

However, the linker can access both the load address and run address of every piece of a split object component. Using the table() operator, you can tell the linker to generate this information into a copy table. The linker gives each piece of the split object component a COPY\_RECORD entry in the copy table object.

For example, consider an application which has 7 tasks. Tasks 1 through 3 are overlaid with tasks 4 through 7 (using a UNION directive). The load placement of all of the tasks is split among 4 different memory areas (LMEM1, LMEM2, LMEM3, and LMEM4). The overlay is defined as part of memory area PMEM. You must move each set of tasks into the overlay at run time before any services from the set are used.

You can use table() operators in combination with splitting operators, >>, to create copy tables that have all the information needed to move either group of tasks into the memory overlay as shown in [Example 7-22](#). [Example 7-23](#) illustrates a possible driver for such an application.

**Example 7-22. Creating a Copy Table to Access a Split Object Component**

```

SECTIONS
{
    UNION
    {
        .task1to3: { *(.task1), *(.task2), *(.task3) }
        load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)

        GROUP
        {
            .task4: { *(.task4) }
            .task5: { *(.task5) }
            .task6: { *(.task6) }
            .task7: { *(.task7) }

        } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)

    } run = PMEM

    ...

    .ovly: > LMEM4
}

```

**Example 7-23. Split Object Component Driver**

```

#include <cpy_tbl.h>

extern far COPY_TABLE task13_ctbl;
extern far COPY_TABLE task47_ctbl;

extern void task1(void);
...
extern void task7(void);

main()
{
    ...
    copy_in(&task13_ctbl);
    task1();
    task2();
    task3();
    ...

    copy_in(&task47_ctbl);
    task4();
    task5();
    task6();
    task7();
    ...
}

```

You must declare a COPY\_TABLE object as *far* to allow the overlay copy table section placement to be independent from the other sections containing data objects (such as .bss).

The contents of the .task1to3 section are split in the section's load space and contiguous in its run space. The linker-generated copy table, \_task13\_ctbl, contains a separate COPY\_RECORD for each piece of the split section .task1to3. When the address of \_task13\_ctbl is passed to copy\_in(), each piece of .task1to3 is copied from its load location into the run location.

The contents of the GROUP containing tasks 4 through 7 are also split in load space. The linker performs the GROUP split by applying the split operator to each member of the GROUP in order. The copy table for the GROUP then contains a COPY\_RECORD entry for every piece of every member of the GROUP. These pieces are copied into the memory overlay when the `_task47_ctbl` is processed by `copy_in()`.

The split operator can be applied to an output section, GROUP, or the load placement of a UNION or UNION member. The linker does not permit a split operator to be applied to the run placement of either a UNION or of a UNION member. The linker detects such violations, emits a warning, and ignores the offending split operator usage.

## 7.16 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as *partial linking* or *incremental linking*. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program.

Follow these guidelines for producing a file that you will relink:

- The intermediate files produced by the linker *must* have relocation information. Use the `-r` option when you link the file the first time. (See [Section 7.4.1.2, Relocation Capabilities \(-a and -r Options\)](#).)
- Intermediate files *must* have symbolic information. By default, the linker retains symbolic information in its output. Do not use the `-s` option if you plan to relink a file, because `-s` strips symbolic information from the output module. (See [Section 7.4.15, Strip Symbolic Information \(-s Option\)](#).)
- Intermediate link steps should be concerned only with the formation of output sections and not with allocation. All allocation, binding, and MEMORY directives should be performed in the final link step.
- If the intermediate files have global symbols that have the same name as global symbols in other files and you want them to be treated as static (visible only within the intermediate file), you must link the files with the `-h` option (see [Section 7.4.9, Make All Global Symbols Static \(-h Option\)](#)).
- If you are linking C code, do not use `-c` or `-cr` until the final link step. Every time you invoke the linker with the `-c` or `-cr` option, the linker attempts to create an entry point. (See [Section 7.4.5, C Language Options \(-c and -cr Options\)](#).)

The following example shows how you can use partial linking:

**Step 1:** Link the file `file1.com`; use the `-r` option to retain relocation information in the output file `tempout1.out`.

```
c16x -z -r -o tempout1 file1.com
```

`file1.com` contains:

```
SECTIONS { ss1: { f1.obj f2.obj . . . fn.obj } }
```

**Step 2:** Link the file `file2.com`; use the `-r` option to retain relocation information in the output file `tempout2.out`.

```
c16x -z -r -o tempout2 file2.com
```

`file2.com` contains:

```
SECTIONS { ss2: { g1.obj g2.obj . . . gn.obj } }
```

**Step 3:** Link `tempout1.out` and `tempout2.out`.

```
c16x -z -m final.map -o final.out tempout1.out tempout2.out
```

## 7.17 Linking C/C++ Code

The C/C++ compiler produces assembly language source code that can be assembled and linked. For example, a C program consisting of modules prog1, prog2, etc., can be assembled and then linked to produce an executable file called prog.out:

```
cl6x -z -c -o prog.out prog1.obj prog2.obj ... rts6200.lib
```

The -c option tells the linker to use special conventions that are defined by the C/C++ environment.

The archive libraries listed below contain C/C++ run-time-support functions:

rts6200.lib	rts6400.lib	rts64plus.lib	rts6700.lib	rts67plus.lib
rts6200_eh.lib	rts6400_eh.lib	rts64plus_eh.lib	rts6700_eh.lib	rts67plus_eh.lib
rts6200e.lib	rts6400e.lib	rts64pluse.lib	rts6700e.lib	rts67pluse.lib
rts6200e_eh.lib	rts6400e_eh.lib	rts64pluse_eh.lib	rts6700e_eh.lib	rts67pluse_eh.lib

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

For more information about the TMS320C6000 C/C++ language, including the run-time environment and run-time-support functions, see the *TMS320C6000 Optimizing Compiler User's Guide*.

### 7.17.1 Run-Time Initialization

All C/C++ programs must be linked with code to initialize and execute the program, called a *bootstrap* routine, also known as the *boot.obj* object module. The symbol `_c_int00` is defined as the program entry point and is the start of the C boot routine in *boot.obj*; referencing `_c_int00` ensures that *boot.obj* is automatically linked in from the run-time-support library. When a program begins running, it executes *boot.obj* first. The *boot.obj* symbol contains code and data for initializing the run-time environment and performs the following tasks:

- Sets up the system stack and configuration registers
- Processes the run-time *.cinit* initialization table and autoinitializes global variables (when the linker is invoked with the -c option)
- Disables interrupts and calls `_main`

The run-time-support object libraries contain *boot.obj*. You can:

- Use the archiver to extract *boot.obj* from the library and then link the module in directly.
- Include the appropriate run-time-support library as an input file (the linker automatically extracts *boot.obj* when you use the -c or -cr option).

### 7.17.2 Object Libraries and Run-Time Support

The *TMS320C6000 Optimizing C/C++ Compiler User's Guide* describes additional run-time-support functions that are included in *rts.src*. If your program uses any of these functions, you must link the appropriate run-time-support library with your object files.

You can also create your own object libraries and link them. The linker includes and links only those library members that resolve undefined references.

### 7.17.3 Setting the Size of the Stack and Heap Sections

The C/C++ language uses two uninitialized sections called *.system* and *.stack* for the memory pool used by the `malloc( )` functions and the run-time stacks, respectively. You can set the size of these by using the -heap or -stack option and specifying the size of the section as a 4-byte constant immediately after the option. The default size for both, if the options are not used, is 1K words.

See [Section 7.4.10, Define Heap Size \(-heap size Option\)](#) and [Section 7.4.16, Define Stack Size \(-stack size Option\)](#) for more information on setting stack sizes.



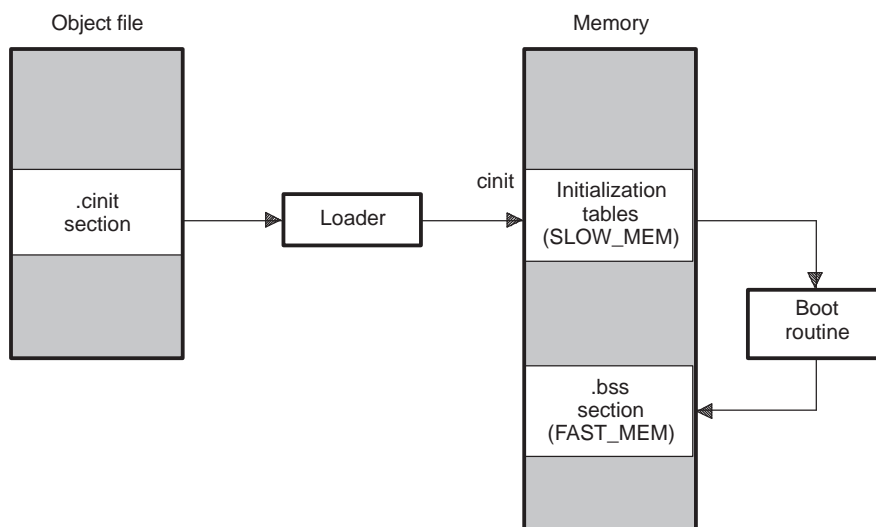
### 7.17.4 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the `-c` option.

Using this method, the `.cinit` section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called `cinit` that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by `.cinit`) into the specified variables in the `.bss` section. This allows initialization data to be stored in slow external memory and copied to fast external memory each time the program starts.

Figure 7-5 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into slow external memory.

Figure 7-5. Autoinitialization at Run Time



### 7.17.5 Initialization of Variables at Load Time

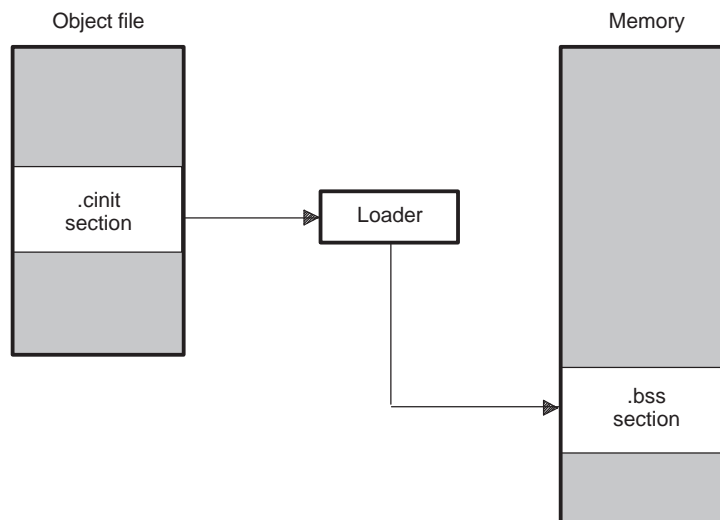
Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the `-cr` option.

When you use the `-cr` linker option, the linker sets the `STYP_COPY` bit in the `.cinit` section's header. This tells the loader not to load the `.cinit` section into memory. (The `.cinit` section occupies no space in the memory map.) The linker also sets the `cinit` symbol to `-1` (normally, `cinit` points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the `.cinit` section in the object file.
- Determine that `STYP_COPY` is set in the `.cinit` section header, so that it knows not to copy the `.cinit` section into memory.
- Understand the format of the initialization tables.

Figure 7-6 illustrates the initialization of variables at load time.

**Figure 7-6. Initialization at Load Time**

### 7.17.6 The -c and -cr Linker Options

The following list outlines what happens when you invoke the linker with the -c or -cr option.

- The symbol `_c_int00` is defined as the program entry point. The `_c_int00` symbol is the start of the C boot routine in `boot.obj`; referencing `_c_int00` ensures that `boot.obj` is automatically linked in from the appropriate run-time-support library.
- The `.cinit` output section is padded with a termination record to designate to the boot routine (autoinitialize at run time) or the loader (initialize at load time) when to stop reading the initialization tables.
- When you autoinitialize at run time (-c option), the linker defines `cinit` as the starting address of the `.cinit` section. The C boot routine uses this symbol as the starting point for autoinitialization.
- When you initialize at load time (-cr option):
  - The linker sets `cinit` to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The `STYP_COPY` flag (0010h) is set in the `.cinit` section header. `STYP_COPY` is the special attribute that tells the loader to perform initialization directly and not to load the `.cinit` section into memory. The linker does not allocate space in memory for the `.cinit` section.

## 7.18 Linker Example

This example links three object files named `demo.obj`, `ctrl.obj`, and `tables.obj` and creates a program called `demo.out`.

Assume that target memory has the following program memory configuration:

Address Range	Contents
0x00000000 to 0x00001000	SLOW_MEM
0x00001000 to 0x00002000	FAST_MEM
0x08000000 to 0x08000400	EEPROM

The output sections are constructed from the following input sections:

- Executable code, contained in the .text sections of demo.obj, ctrl.obj, and tables.obj, must be linked into FAST\_MEM.
- A set of interrupt vectors, contained in the .intvecs section of tables.obj, must be linked at address 0x00000000.
- A table of coefficients, contained in the .data section of tables.obj, must be linked into EEPROM. The remainder of block EEPROM must be initialized to the value 0xFF00FF00.
- A set of variables, contained in the .bss section of ctrl.obj, must be linked into SLOW\_MEM and preinitialized to 0x00000100.
- The .bss sections of demo.obj and tables.obj must be linked into SLOW\_MEM.

Example 7-24 shows the linker command file for this example. Example 7-25 shows the map file.

#### Example 7-24. Linker Command File, demo.cmd

```

/*****
Specify Linker Options
*****/
/*****
-e SETUP          /* Define the program entry point */
-o demo.out        /* Name the output file */
-m demo.map        /* Create an output map */
*****/
/*****
Specify the Input Files
*****/
demo.objctrl.objtables.obj
/*****
Specify the Memory Configuration
*****/
MEMORY
{
    FAST_MEM : org = 0x00000000 len = 0x00001000
    SLOW_MEM : org = 0x00001000 len = 0x00001000
    EEPROM   : org = 0x08000000 len = 0x00000400
}
/*****
Specify the Output Sections
*****/
SECTIONS
{
    .text : {} > FAST_MEM /* Link all .text sections into ROM */
    .intvecs : {} > 0x0 /* Link interrupt vectors at 0x0 */
    .data : /* Link .data sections */
    {
        tables.obj(.data)
        . = 0x400; /* Create hole at end of block */
    } = 0xFF00FF00 > EEPROM /* Fill and link into EEPROM */
    ctrl_vars: /* Create new ctrl_vars section */
    {
        ctrl.obj(.bss)
    } = 0x00000100 > SLOW_MEM /* Fill with 0x100 and link into RAM */
    .bss : {} > SLOW_MEM /* Link remaining .bss sections into RAM */
}
/*****
End of Command File
*****/

```

Invoke the linker by entering the following command:

```
cl6x -z demo.cmd
```

*Linker Example*

This creates the map file shown in [Example 7-25](#) and an output file called demo.out that can be run on a TMS320C6000.

**Example 7-25. Output Map File, demo.map**

```

OUTPUT FILE NAME:    <demo.out>
ENTRY POINT SYMBOL:  0

MEMORY CONFIGURATION

      name      origin      length      used      attributes      fill
      -----
FAST_MEM  00000000  000001000  00000078      RWIX
SLOW_MEM  00001000  000001000  00000502      RWIX
EEPROM    08000000  000000400  00000400      RWIX

SECTION ALLOCATION MAP

  output
  section  page      origin      length      attributes/
  -----
.text      0        00000000  00000064
           00000000  00000030      demo.obj (.text)
           00000030  00000000      tables.obj (.text)
           00000030  00000010      --HOLE-- [fill = 00000000]
           00000040  00000024      ctrl.obj (.text)
.intvecs   0        00000000  00000014
           00000000  00000014      tables.obj (.intvecs)
.data      0        08000000  00000400
           08000000  00000004      tables.obj (.data)
           08000004  000003fc      --HOLE-- [fill = ff00ff00]
           08000400  00000000      ctrl.obj (.data)
           08000400  00000000      demo.obj (.data)
ctrl_vars  0        00001000  00000500
           00001000  00000500      ctrl.obj (.bss) [fill = 00000100]
.bss       0        00001500  00000002      UNINITIALIZED
           00001500  00000002      demo.obj (.bss)
           00001502  00000000      tables.obj (.bss)

GLOBAL SYMBOLS

address  name
-----
00001500 $bss
00001500 .bss
08000000 .data
00000000 .text
00000018 _SETUP
00000040 _fill_tab
00000000 _x42
08000400 edata
00001502 end
00000064 etext
08000000 gvar
08000000 edata
[11 symbols]

address  name
-----
00000000 .text
00000000 _x42
00000018 _SETUP
00000040 _fill_tab
00000064 etext
00001500 $bss
00001500 .bss
00001502 end
08000000 gvar
08000000 .data
08000400 edata

```

## **Absolute Lister Description**

The TMS320C6000™ absolute lister is a debugging tool that accepts linked object files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code. Manually, this could be a tedious process requiring many operations; however, the absolute lister utility performs these operations automatically.

---

### **Absolute Listing Is Not Supported for C6400+**

**Note:** The absolute listing capability is not supported for C6400+. You can use the disassembler (dis6x) or the -m linker option instead.

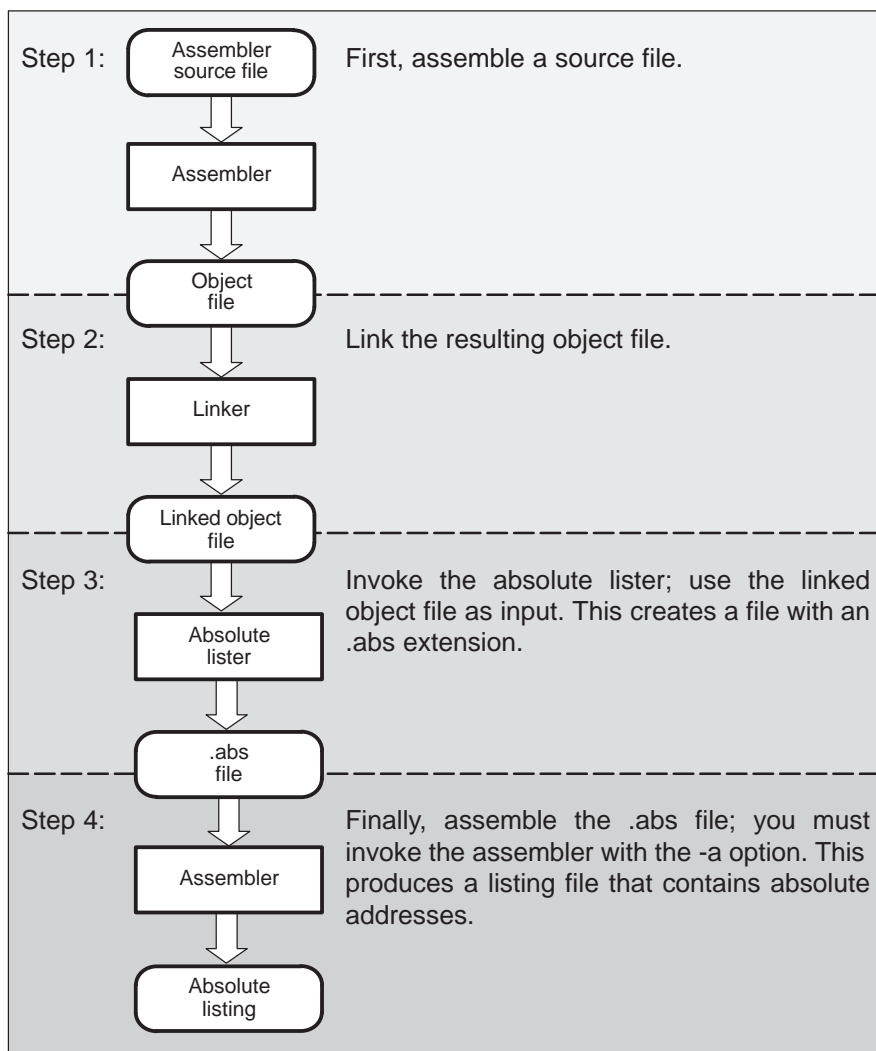
---

Topic	Page
<b>8.1 Producing an Absolute Listing .....</b>	<b>214</b>
<b>8.2 Invoking the Absolute Lister.....</b>	<b>215</b>
<b>8.3 Absolute Lister Example .....</b>	<b>216</b>

## 8.1 Producing an Absolute Listing

Figure 8-1 illustrates the steps required to produce an absolute listing.

**Figure 8-1. Absolute Lister Development Flow**



## 8.2 Invoking the Absolute Lister

The syntax for invoking the absolute lister is:

**abs6x** [-options] *input file*

<b>abs6x</b>	is the command that invokes the absolute lister.
<i>options</i>	identifies the absolute lister options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen (-). The absolute lister options are as follows: <ul style="list-style-type: none"> <li><b>-e</b> enables you to change the default naming conventions for filename extensions on assembly files, C source files, and C header files. The three options are listed below. <ul style="list-style-type: none"> <li>ea [.]<i>asmext</i> for assembly files (default is .asm)</li> <li>ec [.]<i>cext</i> for C source files (default is .c)</li> <li>eh [.]<i>hext</i> for C header files (default is .h)</li> </ul> <p>The . in the extensions and the space between the option and the extension are optional.</p> </li> <li><b>-q</b> (quiet) suppresses the banner and all progress information.</li> </ul>
<i>input file</i>	names the linked object file. If you do not supply an extension, the absolute lister assumes that the input file has the default extension .out. If you do not supply an input filename when you invoke the absolute lister, the absolute lister prompts you for one.

The absolute lister produces an output file for each file that was linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

Assemble these files with the -aa assembler option as follows to create the absolute listing:

**cl6x -aa filename .abs**

The -e options affect both the interpretation of filenames on the command line and the names of the output files. They should always precede any filename on the command line.

The -e options are useful when the linked object file was created from C files compiled with the debugging option (-g compiler option). When the debugging option is set, the resulting linked object file contains the name of the source files used to build it. In this case, the absolute lister does not generate a corresponding .abs file for the C header files. Also, the .abs file corresponding to a C source file uses the assembly file generated from the C source file rather than the C source file itself.

For example, suppose the C source file hello.csr is compiled with the debugging option set; the debugging option generates the assembly file hello.s. The hello.csr file includes hello.hsr. Assuming the executable file created is called hello.out, the following command generates the proper .abs file:

```
abs6x -ea s -ec csr -eh hsr hello.out
```

An .abs file is not created for hello.hsr (the header file), and hello.abs includes the assembly file hello.s, not the C source file hello.csr.

## Absolute Lister Example

### 8.3 Absolute Lister Example

This example uses three source files. The files module1.asm and module2.asm both include the file globals.def.

#### module1.asm

```
.text
.align 4
.bss array, 100
.bss dflag, 4
.copy globals.def

MVKL offset, A0
MVKH offset, A0
LDW  *+b14(dflag), A2
nop 4
```

#### module2.asm

```
.bss offset,2
.copy globals.def

mvkl offset,a0
mvkh offset,a0
mvkl array,a3
mvkh array,a3
```

#### globals.def

```
.global dflag
.global array
.global offset
```

The following steps create absolute listings for the files module1.asm and module2.asm:

**Step 1:** First, assemble module1.asm and module2.asm:

```
c16x module1 c16x module2
```

This creates two object files called module1.obj and module2.obj.

**Step 2:** Next, link module1.obj and module2.obj using the following linker command file, called bttest.cmd:

```
-o bttest.out
-m bttest.map
module1.obj
module2.obj
MEMORY
{
    PMEM: origin=00000000h length=00010000h
    DMEM: origin=80000000h length=00010000h
}
SECTIONS
{
    .data: >DMEM
    .text: >PMEM
    .bss: >DMEM
}
```

Invoke the linker:

```
c16x -z bttest.cmd
```

This command creates an executable object file called bttest.out; use this new file as input for the absolute lister.



**Step 3:** Now, invoke the absolute lister:

```
abs6x bttest.out
```

This command creates two files called module1.abs and module2.abs:

**module1.abs:**

```
.nolist
array .setsym 080000000h
dflag .setsym 080000064h
offset .setsym 080000068h
.data .setsym 080000000h
__data__ .setsym 080000000h
edata .setsym 080000000h
__edata__ .setsym 080000000h
.text .setsym 000000000h
__text__ .setsym 000000000h
etext .setsym 000000040h
__etext__ .setsym 000000040h
.bss .setsym 080000000h
__bss__ .setsym 080000000h
end .setsym 08000006ah
__end__ .setsym 08000006ah
$bss .setsym 080000000h
.setsect ".text",000000020h
.setsect ".data",080000000h
.setsect ".bss",080000000h
.list
.text
.copy "module1.asm"
```

**module2.abs:**

```
.nolist
array .setsym 080000000h
dflag .setsym 080000064h
offset .setsym 080000068h
.data .setsym 080000000h
__data__ .setsym 080000000h
edata .setsym 080000000h
__edata__ .setsym 080000000h
.text .setsym 000000000h
__text__ .setsym 000000000h
etext .setsym 000000040h
__etext__ .setsym 000000040h
.bss .setsym 080000000h
__bss__ .setsym 080000000h
end .setsym 08000006ah
__end__ .setsym 08000006ah
$bss .setsym 080000000h
.setsect ".text",000000000h
.setsect ".data",080000000h
.setsect ".bss",080000068h
.list
.text
.copy "module2.asm"
```

These files contain the following information that the assembler needs for step 4:

- They contain .setsym directives, which equate values to global symbols. Both files contain global equates for the symbol *dflag*. The symbol *dflag* was defined in the file *globals.def*, which was included in *module1.asm* and *module2.asm*.
- They contain .setsect directives, which define the absolute addresses for sections.
- They contain .copy directives, which defines the assembly language source file to include.

The .setsym and .setsect directives are useful only for creating absolute listings, not normal assembly.

*Absolute Lister Example*

**Step 4:** Finally, assemble the .abs files created by the absolute lister (remember that you must use the -aa option when you invoke the assembler):

```
cl6x -aa module1.abs cl6x -aa module2.abs
```

This command sequence creates two listing files called module1.lst and module2.lst; no object code is produced. These listing files are similar to normal listing files; however, the addresses shown are absolute addresses.

The absolute listing files created are module1.lst (see [Example 8-1](#) ) and module2.lst (see [Example 8-2](#)).

**Example 8-1. module1.lst**

```
TMS320C6x COFF Assembler          Version x.xx          Mon Jan  5 11:34:00 1998
Copyright (c) 1996-1998 Texas Instruments Incorporated
module1.abs                                PAGE      1
    22 00000020                        .text
    23                                .copy      "module1.asm"
A    1 00000020                        .text
A    2                                .align    4
A    3 80000000                        .bss      array, 100
A    4 80000064                        .bss      dflag, 4
A    5                                .copy    globals.def
B    1                                .global  dflag
B    2                                .global  array
B    3                                .global  offset
A    6
A    7 00000020 00003428!             MVKL      offset, A0
A    8 00000024 00400068!             MVKH      offset, A0
A    9 00000028 0100196C-             LDW       *+b14(dflag), A2
A   10 0000002c 00006000             nop        4
No Errors, No Warnings
```

**Example 8-2. module2.lst**

```
TMS320C6x COFF Assembler          Version x.xx          Mon Jan  5 11:34:05 1998
Copyright (c) 1996-1998 Texas Instruments Incorporated
module2.abs                                PAGE      1
    22 00000000                        .text
    23                                .copy      "module2.asm"
A    1 80000068                        .bss      offset,2
A    2                                .copy    globals.def
B    1                                .global  dflag
B    2                                .global  array
B    3                                .global  offset
A    3
A    4 00000000 00003428-             mvkl      offset,a0
A    5 00000004 00400068-             mvkh      offset,a0
A    6 00000008 01800028!             mvkl      array,a3
A    7 0000000c 01C00068!             mvkh      array,a3
No Errors, No Warnings
```

## Cross-Reference Lister Description

---

The TMS320C6000™ cross-reference lister is a debugging tool. This utility accepts linked object files as input and produces a cross-reference listing as output. This listing shows symbols, their definitions, and their references in the linked source files.

---

**Cross-Reference Listing Not Supported for C6400+**

**Note:** The cross-reference listing capability is not supported for C6400+. You can use the disassembler, the -m linker option or the object file utility (ofd6x) to obtain similar information.

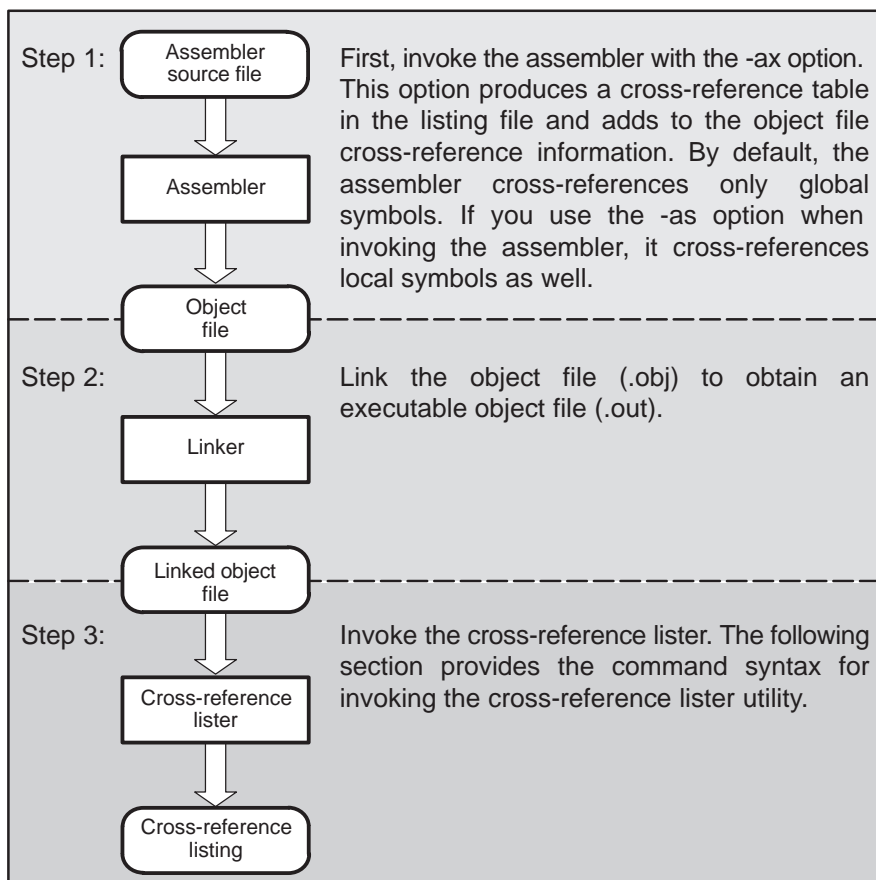
---

Topic	Page
9.1 Producing a Cross-Reference Listing .....	<a href="#">220</a>
9.2 Invoking the Cross-Reference Lister .....	<a href="#">220</a>
9.3 Cross-Reference Listing Example .....	<a href="#">221</a>

## 9.1 Producing a Cross-Reference Listing

Figure 9-1 illustrates the steps required to produce a cross-reference listing.

**Figure 9-1. The Cross-Reference Lister in the TMS320C6000 Software Development Flow**



## 9.2 Invoking the Cross-Reference Lister

To use the cross-reference utility, the file must be assembled with the correct options and then linked into an executable file. Assemble the assembly language files with the -ax option. This option creates a cross-reference listing and adds cross-reference information to the object file. By default, the assembler cross-references only global symbols, but if the assembler is invoked with the -as option, local symbols are also added. Link the object files to obtain an executable file.

To invoke the cross-reference lister, enter the following:

**xref6x** [*options*] [*input filename*] [*output filename*]]

**xref6x** is the command that invokes the cross-reference utility.

*options* identifies the cross-reference lister options you want to use. Options are not case sensitive and can appear anywhere on the command line following the command.

**-l** (lowercase L) specifies the number of lines per page for the output file. The format of the -l option is *-lnum*, where num is a decimal constant. For example, -l30 sets the number of lines per page in the output file to 30. The space between the option and the decimal constant is optional. The default is 60 lines per page.

**-q** suppresses the banner and all progress information (run quiet).

*input filename* is a linked object file. If you omit the input filename, the utility prompts for a filename.

*output filename* is the name of the cross-reference listing file. If you omit the output filename, the default filename is the input filename with an .xrf extension.

## 9.3 Cross-Reference Listing Example

[Example 9-1](#) is an example of cross-reference listing.

### Example 9-1. Cross-Reference Listing

=====							
Symbol: _SETUP							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
-----							
demo.asm	EDEF	'00000018	00000018	18	13	20	
=====							
Symbol: _fill_tab							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
-----							
ctrl.asm	EDEF	'00000000	00000040	10	5		
=====							
Symbol: _x42							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
-----							
demo.asm	EDEF	'00000000	00000000	7	4	18	
=====							
Symbol: gvar							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
-----							
tables.asm	EDEF	"00000000	08000000	11	10		
=====							

*Cross-Reference Listing Example*

The terms defined below appear in the preceding cross-reference listing:

<b>Symbol</b>	Name of the symbol listed
<b>Filename</b>	Name of the file where the symbol appears
<b>RTYP</b>	The symbol's reference type in this file. The possible reference types are: <b>STAT</b> The symbol is defined in this file and is not declared as global. <b>EDEF</b> The symbol is defined in this file and is declared as global. <b>EREF</b> The symbol is not defined in this file but is referenced as global. <b>UNDF</b> The symbol is not defined in this file and is not declared as global.
<b>AsmVal</b>	This hexadecimal number is the value assigned to the symbol at assembly time. A value may also be preceded by a character that describes the symbol's attributes. <a href="#">Table 9-1</a> lists these characters and names.
<b>LnkVal</b>	This hexadecimal number is the value assigned to the symbol after linking.
<b>DefLn</b>	The statement number where the symbol is defined.
<b>RefLn</b>	The line number where the symbol is referenced. If the line number is followed by an asterisk (*), then that reference can modify the contents of the object. A blank in this column indicates that the symbol was never used.

**Table 9-1. Symbol Attributes in Cross-Reference Listing**

Character	Meaning
'	Symbol defined in a .text section
"	Symbol defined in a .data section
+	Symbol defined in a .sect section
-	Symbol defined in a .bss or .usect section

## Object File Utilities Descriptions

This chapter describes how to invoke the following miscellaneous utilities:

- The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both text and XML formats.
- The **disassembler** writes the disassembled object code from object or executable files.
- The **name utility** prints a list of names defined and referenced in a COFF object or an executable file.
- The **strip utility** removes symbol table and debugging information from object and executable files.

Topic	Page
10.1 Invoking the Object File Display Utility .....	224
10.2 XML Tag Index .....	224
10.3 Example XML Consumer .....	228
10.4 Invoking the Disassembler .....	234
10.5 Invoking the Name Utility .....	235
10.6 Invoking the Strip Utility .....	235

## 10.1 Invoking the Object File Display Utility

The object file display utility, *ofd6x*, prints the contents of object files (.obj), executable files (.out), and/or archive libraries (.lib) in both text and XML formats.

To invoke the object file display utility, enter the following:

**ofd6x** [*options*] *input filenames* [*input filenames*]

- ofd6x** is the command that invokes the object file display utility.
- input filenames* names the assembly language source file. The filename must contain an .asm extension.
- options* identify the object file display utility options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen.
- g** Appends DWARF debug information to program output.
  - o filename** Sends program output to *filename* rather than to the screen.
  - x** Displays output in XML format.

If the object file display utility is invoked without any options, it displays information about the contents of the input files on the console screen.

---

### Object File Display Format

**Note:** The object file display utility produces data in a text format by default. This data is not intended to be used as machine or software input.

---

## 10.2 XML Tag Index

Table 10-1 describes the XML tags that are generated by the object file display utility when invoked with the -x option.

**Table 10-1. XML Tag Index**

Tag Name	Context	Description
<addr>	<line_entry>	PC address
	<row>	PC address
	<value>	Machine address
<addr_class>	<value>	Address class
<addr_size>	<compile_unit>	Size of one machine address (octets)
	<section>	Size of one machine address (octets)
<alignment>	<section>	Alignment factor
<archive>	<ofd>	Archive file (.lib)
<attribute>	<die>	Attribute of a DWARF DIE
<aux_count>	<symbol>	Number of auxiliary entries for this symbol
<banner>	<ofd>	Tool name and version information
<block>	<section>	True if alignment is used as blocking factor
	<value>	Data block
<bss>	<section>	True if this section contains uninitialized data
<bss_size>	<optional_file_header>	Size of uninitialized data



**Table 10-1. XML Tag Index (continued)**

Tag Name	Context	Description
<byte_swapped>	<file_header>	Endianness of build host is opposite of current host
<clink>	<section>	True if this section is conditionally linked
<column>	<line_entry>	Source column number
<compile_unit>	<section>	Compile unit
<const>	<value>	Constant
<copy>	<section>	True if this section is a copy section
<copyright>	<ofd>	Copyright notice
<cpu_flags>	<file_header>	CPU flags
<data>	<section>	True if this section contains initialized data
<data_size>	<optional_file_header>	Size of initialized data
<data_start>	<optional_file_header>	Beginning address of initialized data
<destination>	<register>	Destination register
<die>	<compile_unit>	DWARF debugging information entry (DIE)
<dim_bound>	<dimension>	Dimension upper-bound
<dim_num>	<dimension>	Dimension number
<dimension>	<symbol>	Array dimension
<disp>	<reloc_entry>	Extra address encoding information
<dummy>	<section>	True if this section is a dummy section
<dwarf>	<ti_coff>	DWARF information
<endian>	<file_header>	Endianness of target machine
<entry_point>	<optional_file_header>	Entry point of executable program
<exec>	<file_header>	True if this file is executable
<fde>	<section>	A DWARF frame description entry (FDE)
<field_size>	<reloc_entry>	Size of the field to relocate
<file_header>	<ti_coff>	COFF file header
<file_length>	<file_header>	Size of this file
<file_name>	<line_entry>	Name of source file
	<symbol>	Name of source file
<file_offsets>	<section>	File offsets associated with this section
<flag>	<value>	Flag
<form>	<attribute>	Attribute form
<frame_size>	<symbol>	Size of function frame
<function>	<line_numbers>	Line number entries for one function
<icode>	<section>	True if this section has I-Code associated with it
<index>	<symbol>	Index of this symbol in the symbol table
<indirect_register>	<memory>	Indirect register used for calculating destination address
<initial_location>	<fde>	Start of function referred to by the FDE
<internal>	<reloc_entry>	True if this relocation is internal
<kind>	<symbol>	Kind of symbol (defined, undefined, absolute, symbolic debug)
<length>	<symbol>	Length of section
<line>	<line_entry>	Source line number
	<symbol>	First source line associated with this symbol
<line_count>	<section>	Number of line number entries
	<symbol>	Number of line number entries
<line_entry>	<compile_unit>	Line number entry
	<line_numbers>	Line number entry

**Table 10-1. XML Tag Index (continued)**

Tag Name	Context	Description
<line_numbers>	<section>	Line number entries
<line_ptr>	<file_offsets>	File offset of line number entries
	<symbol>	File offset of line number entries
<Inno_strip>	<file_header>	True if line numbers were stripped from this file
<localsym_strip>	<file_header>	True if local symbols were stripped from this file
<magic>	<optional_file_header>	Optional file header magic number (0x0108)
<math_relative>	<reloc_entry>	True if this relocation is math relative
<memory>	<row>	SOE register is saved to memory
<name>	<fde>	Name of function referred to by the FDE
	<function>	Name of the current function
	<ofd>	Name of an object or archive file
	<section>	Name of this section
	<symbol>	Name of this symbol
<next_symbol>	<symbol>	Index of next symbol after multi-symbol entity
<noload>	<section>	True if this section is a no-load section
<object_file>	<ofd>	Object file (.obj, .out)
<ofd>		Object file display (OFD) document
<offset>	<memory>	Offset of destination address from indirect register
	<reloc_entry>	Offset of the field from relocatable address
<optional_file_header>	<ti_coff>	Optional file header
<padded>	<section>	True if this section has been padded (C55x only)
<page>	<section>	Memory page
<pass>	<section>	True if this section is passed through unchanged
<physical_addr>	<section>	Physical (run) address of section
<raw_data_ptr>	<file_offsets>	File offset of raw data
<raw_data_size>	<section>	Size of raw data (octets)
<ref>	<value>	Reference
<register>	<row>	SOE register is saved to register
<register_mask>	<symbol>	Mask of saved SOE registers
<regular>	<section>	True if this section is a regular section
<reloc_count>	<section>	Number of relocation entries
	<symbol>	Number of relocation entries
<reloc_entry>	<relocations>	Relocation entry
<reloc_ptr>	<file_offsets>	File offset of relocation entries
<reloc_strip>	<file_header>	True if relocation information was stripped from this file
<relocations>	<section>	Relocation entries
<return_address_register>	<fde>	Register used to pass the return address of this function
<row>	<table>	Table row
<section>	<dwarf>	DWARF section
	<symbol>	Section containing the definition of this symbol
	<ti_coff>	COFF section
<section_count>	<file_header>	Number of section headers
<size_in_addrs>	<symbol>	Number of machine-address-sized units in function
<size_in_bits>	<symbol>	Size of symbol (bits)
<source>	<memory>	Source register
	<register>	Source register

**Table 10-1. XML Tag Index (continued)**

Tag Name	Context	Description
<start_symbol>	<symbol>	First symbol in multi-symbol entity
<storage_class>	<symbol>	Storage class of this symbol
<storage_type>	<symbol>	Storage type of this symbol
<string>	<string_table> <value>	String table entry String
<string_table>	<ti_coff>	String table
<string_table_size>	<string_table>	Size of string table
<sym_merge>	<file_header>	True if debug type-symbols were merged
<symbol>	<symbol_table>	Symbol table entry
<symbol_count>	<file_header>	Number of entries in the symbol table
<symbol_relative>	<reloc_entry>	Relocation is relative to the specified symbol
<symbol_table>	<ti_coff>	Symbol table
<table>	<fde>	FDE table
<tag>	<die>	Tag name
<tag_index>	<symbol>	Reference to user-defined type
<target_id>	<file_header>	Target ID; magic number identifying the target machine
<text>	<section>	True if this section contains code
<text_size>	<optional_file_header>	Size of executable code
<text_start>	<optional_file_header>	Beginning address of executable code
<ti_coff>	<object_file>	TI COFF file
<tool_version>	<optional_file_header>	Tool version stamp
<type>	<attribute> <reloc_entry>	Attribute type Type of relocation
<type_ref>	<value>	Type reference
<value>	<attribute> <reloc_entry> <symbol>	Attribute value Value Value
<vector>	<section>	True if this section contains a vector table (C55x only)
<version>	<compile_unit> <file_header>	DWARF version Version ID; structure version of this COFF file
<virtual_addr>	<reloc_entry> <section>	Virtual address to be relocated Virtual (load) address of section
<word_size>	<reloc_entry>	Number of address-sized units containing the relocation field
<xml_version>	<dwarf> <ti_coff>	Version of the DWARF XML language Version of the COFF XML language

## 10.3 Example XML Consumer

In this section, we present an example of a small application that uses the XML output of ofd6x to calculate the size of the executable code contained in an object file.

The example contains three source files: codesize.cpp, xml.h, and xml.cpp. When compiled into an executable named codesize, it can be used with ofd6x from the command line as follows:

```
% ofd6x -x a.out | codesize

Code Section Name: .text
Code Section Size: 44736

Code Section Name: .text2
Code Section Size: 64

Code Section Name: .text3
Code Section Size: 64

Total Code Size: 44864
```

### 10.3.1 The Main Application

The codesize.cpp file contains the main application for the object file display utility example.

```
//*****
// CODESIZE.CPP - An example application that calculates the size of the
// executable code in an object file using the XML output
// of the OFD utility.
//*****
#include "xml.h"
#include <iostream>

using namespace std;

static void parse_XML_prolog(istream &in);

//*****
// main() - List the names and sizes of the code sections (in octets), and
//          output the total code size.
//*****
int main()
{
    //-----
    // Build our tree of XML Entities from standard input (See xml.{cpp,h} for -
    // the definition of the XMLEntity object).
    //-----
    parse_XML_prolog(cin);
    XMLEntity *root = new XMLEntity(cin);
    //-----
    // Fetch the XML Entities of the section roots. In other words, get a -
    // list of all the XMLEntity sub-trees named "section" that are in the -
    // context of "ofd->object_file->ti_coff", where "ofd" is the root of our -
    // XML document.
    //-----
    CEntityList query_result;
    const char *section_query[] =
        { "ofd", "object_file", "ti_coff", "section", NULL };

    query_result = root->query(section_query);

    //-----
    // Iterate over the section Entities, looking for code sections.
    //-----
    CEntityList_CIt pit;
    unsigned long total_code_size = 0;
```

```

for (pit = query_result.begin(); pit != query_result.end(); ++pit)
{
    //-----
    // Query for the name, text, and raw_data_size sub-entities of each -
    // section. XMLEntity::query() always returns a list, even if there -
    // will only ever be a maximum of one result. If the tag is not -
    // found, an empty list is returned. -
    //-----
    const char *section_name_query[] = { "section", "name",          NULL };
    const char *section_text_query[] = { "section", "text",          NULL };
    const char *section_size_query[] = { "section", "raw_data_size", NULL };

    CEntityList sname_l;
    CEntityList stext_l;
    CEntityList ssize_l;

    sname_l = (*pit)->query(section_name_query);
    stext_l = (*pit)->query(section_text_query);
    ssize_l = (*pit)->query(section_size_query);
    //-----
    // If a "text" flag was found, this is a code section. Output -
    // the section name and size, and add its size to our total code size -
    // counter. -
    //-----
    if (stext_l.size() > 0)
    {
        unsigned long size;

        size = strtoul((*ssize_l.begin()->value()).c_str(), NULL, 16);

        cout << "Code Section Name:  " << (*sname_l.begin()->value()) << endl;
        cout << "Code Section Size:  " << size << endl;
        cout << endl;

        total_code_size += size;
    }
}

//-----
// Output the total code size, and clean up. -
//-----
cout << "Total Code Size:  " << total_code_size << endl;
delete root;

return 0;
}

//*****
// parse_XML_prolog() - Parse the XML prolog, and throw it away. *
//*****
static void parse_XML_prolog(istream &in)
{
    char c;

    while (true)
    {
        //-----
        // Look for the next tag; if it is not an XML directive, we're done. -
        //-----
        for (in.get(c); c != '<' && !in.eof(); in.get(c))
            ; // empty body

        if (in.eof()) return;
        if (in.peek() != '?') { in.unget(); return; }
    }
}

```

### Example XML Consumer

---

```
//-----  
// Otherwise, read in the directive and continue. -  
//-----  
for (in.get(c); c != '>' && !in.eof(); in.get(c))  
    ; // empty body  
}  
}
```

### 10.3.2 xml.h Declaration of the XMLEntity Object

The xml.h file contains the declaration of the XMLEntity object for the codesize.cpp application.

```
//*****
// XML.H - Declaration of the XMLEntity object. *
//*****
#ifndef XML_H
#define XML_H
#include <list>
#include <string>

//*****
// Type Declarations. *
//*****
class XMLEntity;
typedef list<XMLEntity*>      EntityList;
typedef list<const XMLEntity*> CEntityList;
typedef CEntityList::const_iterator CEntityList_CIt;
typedef EntityList::const_iterator EntityList_CIt;
8
//*****
// CLASS XMLENTITY - A Simplified XML Entity Object. *
//*****
class XMLEntity
{
public:
    XMLEntity (istream &in, XMLEntity *parent=NULL);
    ~XMLEntity ();
    const CEntityList query (const char **context) const;
    const string      &tag   () const { return tag_m;      }
    const string      &value () const { return value_m;     }

private:
    void parse_raw_tag (const string &raw_tag);
    void sub_query     (CEntityList &result, const char **context) const;

    string      tag_m;      // Tag Name
    string      value_m;    // Value
    XMLEntity *parent_m;    // Pointer to parent in XML hierarchy
    EntityList children_m;  // List of children in XML hierarchy
};
#endif
```

### 10.3.3 xml.cpp Definition of the XMLEntity Object

The xml.cpp file contains the definition of the XMLEntity object for the codesize.cpp application.

```
//*****
// XML.CPP - Definition of the XMLEntity object. *
//*****
#include "xml.h"
#include <iostream>
#include <string>
#include <list>
#include <cstdlib>

//*****
// XMLEntity::query() - Return the list of XMLEntities a list that reside *
// in the given XML context. *
//*****
const CEntityList XMLEntity::query(const char **context) const
{
    CEntityList result;

    if (!*context) return result;
```

*Example XML Consumer*

```

    sub_query(result, context);

    return result;
}

//*****
// XMLEntity::sub_query() - Recurse through the XML tree looking for a match *
//                          to the current query. *
//*****
void XMLEntity::sub_query(CEntityList &result, const char **context) const
{
    if (!context[0] || tag() != context[0]) return;

    if (!context[1])
        result.push_front(this);
    else
    {
        EntityList_CIt pit;

        for (pit = children_m.begin(); pit != children_m.end(); ++pit)
            (*pit)->sub_query(result, context+1);
    }
    return;
}

//*****
// XMLEntity::parse_raw_tag() - Cut out the tag name from the complete string *
// we found between the < > brackets. This throws out any attributes. *
//*****
void XMLEntity::parse_raw_tag(const string &raw_tag)
{
    {
        string attribute;
        int I;

        for (I = 0; I < raw_tag.size() && raw_tag[I] != ' '; ++I)
            tag_m += raw_tag[I];
    }
}

//*****
// XMLEntity::XMLEntity() - Recursively construct a tree of XMLEntities from *
// the given input stream. *
//*****
XMLEntity::XMLEntity(istream &in, XMLEntity *parent) :
tag_m(""), value_m(""), parent_m(parent)
{
    string raw_tag;
    char c;
    int I;
    //-----
    // Read in the leading '<'.
    //-----
    in.get();

    //-----
    // Store the tag name and attributes in "raw_tag", then call
    // process_raw_tag() to separate the tag name from the attributes and
    // store it in tag_m.
    //-----
    for (in.get(c); c != '>' && c != '/' && !in.eof(); in.get(c))
        raw_tag += c;

    parse_raw_tag(raw_tag);
}

```



```

//-----
// If we're reading in an end-tag, read in the closing '>' and return.      -
//-----
if (c == '/') { in.get(c); return; }

//-----
// Otherwise, parse our value.                                              -
//-----
while (true)
{
    //-----
    // Read in the closing '>', then start reading in characters and add      -
    // them to value_m. Stop when we hit the beginning of a tag.              -
    //-----
    for (in.get(c); c != '<'; in.get(c)) value_m += c;

    //-----
    // If we're reading in a start tag, parse in the entire entity, and        -
    // add it to our child list (recursive constructor call).                  -
    //-----
    if (in.peek() != '/')
    {
        //-----
        // Put back the opening '<', since XMLEntity() expects to read it.      -
        //-----
        in.unget();
        children_m.push_front(new XMLEntity(in, this));
    }
    //-----
    // Otherwise, read in our end tag, and exit.                              -
    //-----
    else
    {
        for (in.get(c); c != '>'; in.get(c))
            ; // empty body
        break;
    }
}

//-----
// Strip off leading and trailing white space from our value.              -
//-----
for (I = 0; I < value_m.size(); I++)
    if (value_m[I] != ' ' && value_m[I] != '\n') break;
value_m.erase(0, I);

for (I = value_m.size()-1; I >= 0; I--)
    if (value_m[I] != ' ' && value_m[I] != '\n') break;
value_m.erase(I+1, value_m.size()-I);
}

//*****
// XMLEntity::~XMLEntity() - Delete a XMLEntity object.                    *
//*****
XMLEntity::~XMLEntity()
{
    EntityList_CIt pit;

    for (pit = children_m.begin(); pit != children_m.end(); ++pit)
        delete (*pit);
}

```

## 10.4 Invoking the Disassembler

The disassembler, *dis6x*, examines the output of the assembler or linker. This utility accepts an object file or executable file as input and writes the disassembled object code to standard output or a specified file.

To invoke the disassembler, enter the following:

**dis6x** *input filename* [*output filename*]

**dis6x** is the command that invokes the disassembler.

*input filename* is a COFF object file (.obj) or an executable file (.out).

*output filename* is the name of the optional output file to which the disassembly will be written. If an output filename is not specified, the disassembly is written to standard output.

When the example file in [Example 10-1](#) is compiled, the assembler will produce an object file, simple.obj.

### Example 10-1. Object File simple.asm

```
.data
coefficients:
    .word 0x11111111
    .word 0x22222222
    .short 0x3333
    .word 0x44444444
    .short 0x5555

.text
foo:
    B.S2 B3
    || MPY.M1X A4, B4, A4
    NOP
    ADD.L1 A4, A6, A4
    NOP 3
```

As shown in [Example 10-2](#), the disassembler can produce disassembly from the object file, simple.obj. The first two lines are entered on the command line.

### Example 10-2. Disassembly From simple.obj

```
% cl6x simple.asm
% dis6x simple.obj

TEXT Section .text (Little Endian), 0x20 bytes at 0x0
00000000      .text:
00000000  000c0363      B.S2      B3
00000004  02109c80 ||      MPY.M1X      A4,B4,A4
00000008  00000000      NOP
0000000c  02188078      ADD.L1      A4,A6,A4
00000010  00004000      NOP      3
00000014  00000000      NOP
00000018  00000000      NOP
0000001c  00000000      NOP

DATA Section .data (Little Endian), 0x12 bytes at 0x0
00000000      .data:
00000000  11111111      .word 0x11111111
00000004  22222222      .word 0x22222222
00000008  00003333      .word 0x00003333
0000000c  44444444      .word 0x44444444
```

### Example 10-2. Disassembly From simple.obj (continued)

```
00000010    00005555                .word 0x00005555
```

## 10.5 Invoking the Name Utility

The name utility, *nm6x*, prints the list of names defined and referenced in a COFF object (.obj) or an executable file (.out). It also prints the symbol value and an indication of the kind of symbol.

To invoke the name utility, enter the following:

```
nm6x [-options] [input filenames]
```

<b>nm6x</b>	is the command that invokes the name utility.
<i>input filename</i>	is a COFF object file (.obj) or an executable file (.out).
<i>options</i>	identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The name utility options are as follows:
<b>-a</b>	prints all symbols.
<b>-c</b>	also prints C_NULL symbols.
<b>-d</b>	also prints debug symbols.
<b>-f</b>	prepends file name to each symbol.
<b>-g</b>	prints only global symbols.
<b>-h</b>	shows the current help screen.
<b>-l</b>	produces a detailed listing of the symbol information.
<b>-n</b>	sorts symbols numerically rather than alphabetically.
<b>-o file</b>	outputs to the given file.
<b>-p</b>	causes the name utility to not sort any symbols.
<b>-q</b>	(quiet mode) suppresses the banner and all progress information.
<b>-r</b>	sorts symbols in reverse order.
<b>-t</b>	also prints tag information symbols.
<b>-u</b>	only prints undefined symbols.

## 10.6 Invoking the Strip Utility

The strip utility, *strip6x*, removes symbol table and debugging information from object and executable files.

To invoke the strip utility, enter the following:

```
strip6x [-p] input filename [input filename]
```

### Invoking the Strip Utility

---

- strip6x** is the command that invokes the strip utility.
- input filename* is a COFF object file (.obj) or an executable file (.out).
- options* identifies the strip utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The strip utility option is as follows:
- p** removes all information not required for execution. This option causes more information to be removed than the default behavior, but the object file is left in a state that cannot be linked. This option should be used only with executable (.out) files.

When the strip utility is invoked, the input object files are replaced with the stripped version.

## Hex Conversion Utility Description

The TMS320C6000™ assembler and linker create object files that are in common object file format (COFF). COFF is a binary object file format that encourages modular programming and provides powerful and flexible methods for managing code segments and target system memory.

Most EPROM programmers do not accept COFF object files as input. The hex conversion utility converts a COFF object file into one of several standard ASCII hexadecimal formats, suitable for loading into an EPROM programmer. The utility is also useful in other applications requiring hexadecimal conversion of a COFF object file (for example, when using debuggers and loaders).

The hex conversion utility can produce these output file formats:

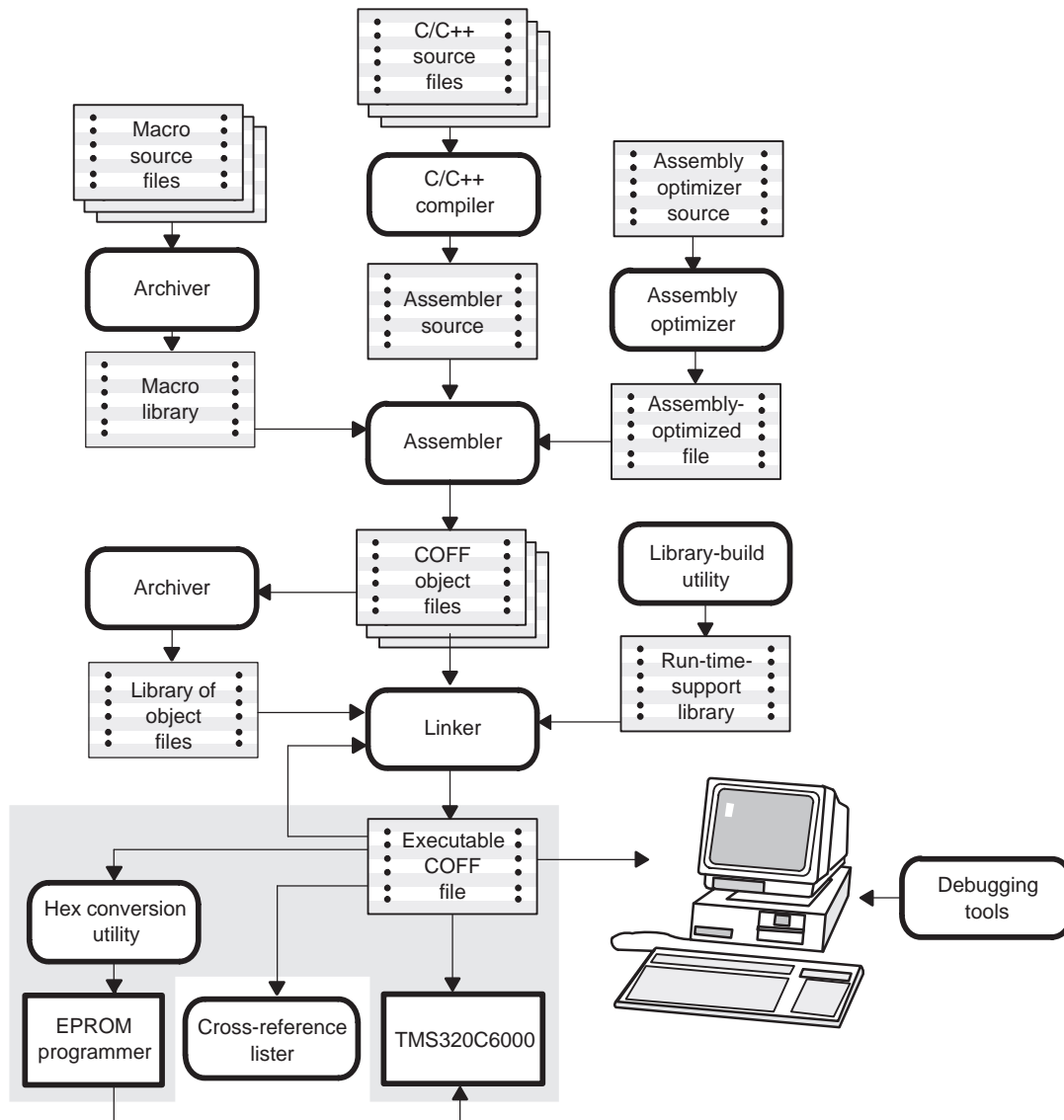
- ASCII-Hex, supporting 16-bit addresses
- Extended Tektronix (Tektronix)
- Intel MCS-86 (Intel)
- Motorola Exorciser (Motorola-S), supporting 16-bit addresses
- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses

Topic	Page
<b>11.1 The Hex Conversion Utility's Role in the Software Development Flow .....</b>	<b>238</b>
<b>11.2 Invoking the Hex Conversion Utility .....</b>	<b>239</b>
<b>11.3 Understanding Memory Widths.....</b>	<b>242</b>
<b>11.4 The ROMS Directive .....</b>	<b>246</b>
<b>11.5 The SECTIONS Directive .....</b>	<b>250</b>
<b>11.6 Excluding a Specified Section .....</b>	<b>251</b>
<b>11.7 Assigning Output Filenames .....</b>	<b>252</b>
<b>11.8 Image Mode and the -fill Option .....</b>	<b>253</b>
<b>11.9 Building a Table for an On-Chip Boot Loader .....</b>	<b>254</b>
<b>11.10 Controlling the ROM Device Address .....</b>	<b>257</b>
<b>11.11 Description of the Object Formats .....</b>	<b>258</b>
<b>11.12 Hex Conversion Utility Error Messages .....</b>	<b>262</b>

## 11.1 The Hex Conversion Utility's Role in the Software Development Flow

Figure 11-1 highlights the role of the hex conversion utility in the software development process.

**Figure 11-1. The Hex Conversion Utility in the TMS320C6000 Software Development Flow**



## 11.2 Invoking the Hex Conversion Utility

There are two basic methods for invoking the hex conversion utility:

- **Specify the options and filenames on the command line.** The following example converts the file `firmware.out` into TI-Tagged format, producing two output files, `firm.lsb` and `firm.msb`.

```
hex6x -t firmware -o firm.lsb -o firm.msb
```

- **Specify the options and filenames in a command file.** You can create a batch file that stores command line options and filenames for invoking the hex conversion utility. The following example invokes the utility using a command file called `hexutil.cmd`:

```
hex6x hexutil.cmd
```

In addition to regular command line information, you can use the hex conversion utility `ROMS` and `SECTIONS` directives in a command file.

### 11.2.1 Invoking the Hex Conversion Utility From the Command Line

To invoke the hex conversion utility, enter:

```
hex6x [options] filename
```

<b>hex6x</b>	is the command that invokes the hex conversion utility.
<b>options</b>	<p>supplies additional information that controls the hex conversion process. You can use options on the command line or in a command file. <a href="#">Table 11-1</a> lists the basic options.</p> <ul style="list-style-type: none"> <li>• All options are preceded by a hyphen and are not case sensitive.</li> <li>• Several options have an additional parameter that must be separated from the option by at least one space.</li> <li>• Options with multicharacter names must be spelled exactly as shown in this document; no abbreviations are allowed.</li> <li>• Options are not affected by the order in which they are used. The exception to this rule is the <code>-q</code> (quiet) option, which must be used before any other options.</li> </ul>
<b>filename</b>	names a COFF object file or a command file (for more information, see <a href="#">Section 11.2.2</a> ). If you do not specify a filename, the utility prompts you for one.

**Table 11-1. Basic Hex Conversion Utility Options**

General Options	Option	Description	See
Control the overall operation of the hex conversion utility.	-exclude <i>section_name</i>	Ignore specified section	<a href="#">Section 11.6</a>
	-map <i>filename</i>	Generate a map file	<a href="#">Section 11.4.2</a>
	-o <i>filename</i>	Specify an output filename	<a href="#">Section 11.7</a>
	-q	Run quietly (when used, it must appear <i>before</i> other options)	<a href="#">Section 11.2.2</a>
Image Options	Option	Description	See
Create a continuous image of a range of target memory	-fill <i>value</i>	Fill holes with <i>value</i>	<a href="#">Section 11.8.2</a>
	-image	Specify image mode	<a href="#">Section 11.8.1</a>
	-zero	Reset the address origin to 0 in image mode	<a href="#">Section 11.8.3</a>
Memory Options	Option	Description	See
Configure the memory widths for your output files	-memwidth <i>value</i>	Define the system memory word width (default 32 bits)	<a href="#">Section 11.3.2</a>
	-romwidth <i>value</i>	Specify the ROM device width (default depends on format used)	<a href="#">Section 11.3.3</a>
	-order L	Output file is in little-endian format	<a href="#">Section 11.3.4</a>
	-order M	Output file is in big-endian format	<a href="#">Section 11.3.4</a>
Output Options	Option	Description	See
Specify the output format	-a	Select ASCII-Hex	<a href="#">Section 11.11.1</a>
	-i	Select Intel	
	-m	Select Motorola-S	<a href="#">Section 11.11.3</a>
	-t	Select TI-Tagged	<a href="#">Section 11.11.4</a>
	-x	Select Tektronix (default)	<a href="#">Section 11.11.5</a>
Boot Options	Option	Description	See
Control the boot loader	-boot	Convert all initialized sections into bootable form (use instead of a SECTIONS directive)	<a href="#">Section 11.9.3.1</a>
	-bootorg	Specify the source address of the boot loader table	<a href="#">Section 11.9.3.1</a>
	-bootsection <i>sectname value</i>	Specify which section contains the boot routine and where it should be placed	<a href="#">Section 11.9.3.1</a>
	-e <i>value</i>	Specify the entry point at which to begin execution after boot loading. The <i>value</i> can be an address or a global symbol.	<a href="#">Section 11.9.3.1</a>



### 11.2.2 Invoking the Hex Conversion Utility With a Command File

A command file is useful if you plan to invoke the utility more than once with the same input files and options. It is also useful if you want to use the ROMS and SECTIONS hex conversion utility directives to customize the conversion process.

Command files are ASCII files that contain one or more of the following:

- **Options and filenames.** These are specified in a command file in exactly the same manner as on the command line.
- **ROMS directive.** The ROMS directive defines the physical memory configuration of your system as a list of address-range parameters. (For more information, see [Section 11.4.](#))
- **SECTIONS directive.** The hex conversion utility SECTIONS directive specifies which sections from the COFF object file are selected. (For more information, see [Section 11.5.](#))
- **Comments.** You can add comments to your command file by using the `/*` and `*/` delimiters. For example:

```
/* This is a comment. */
```

To invoke the utility and use the options you defined in a command file, enter:

**hex6x** *command\_filename*

You can also specify other options and files on the command line. For example, you could invoke the utility by using both a command file and command line options:

```
hex6x firmware.cmd -map firmware.mxp
```

The order in which these options and filenames appear is not important. The utility reads all input from the command line and all information from the command file before starting the conversion process. However, if you are using the `-q` option, *it must appear as the first option on the command line or in a command file.*

The **-q option** suppresses the hex conversion utility's normal banner and progress information.

- Assume that a command file named `firmware.cmd` contains these lines:

```
firmware.out /* input file */
-t           /* TI-Tagged */
-o firm.lsb  /* output file */
-o firm.msb  /* output file */
```

You can invoke the hex conversion utility by entering:

```
hex6x firmware.cmd
```

- This example shows how to convert a file called `appl.out` into eight hex files in Intel format. Each output file is one byte wide and 4K bytes long.

```
appl.out      /* input file */
-i            /* Intel format */
-map appl.mxp /* map file */
```

```
ROMS
{
  ROW1: origin=0x00000000 len=0x4000 romwidth=8
        files={ appl.u0 appl.u1 appl.u2 appl.u3 }
  ROW2: origin=0x00004000 len=0x4000 romwidth=8
        files={ appl.u4 appl.u5 appl.u6 appl.u7 }
}
```

```
SECTIONS
{
  .text, .data, .cinit, .sect1, .vectors, .const:
}
```

### 11.3 Understanding Memory Widths

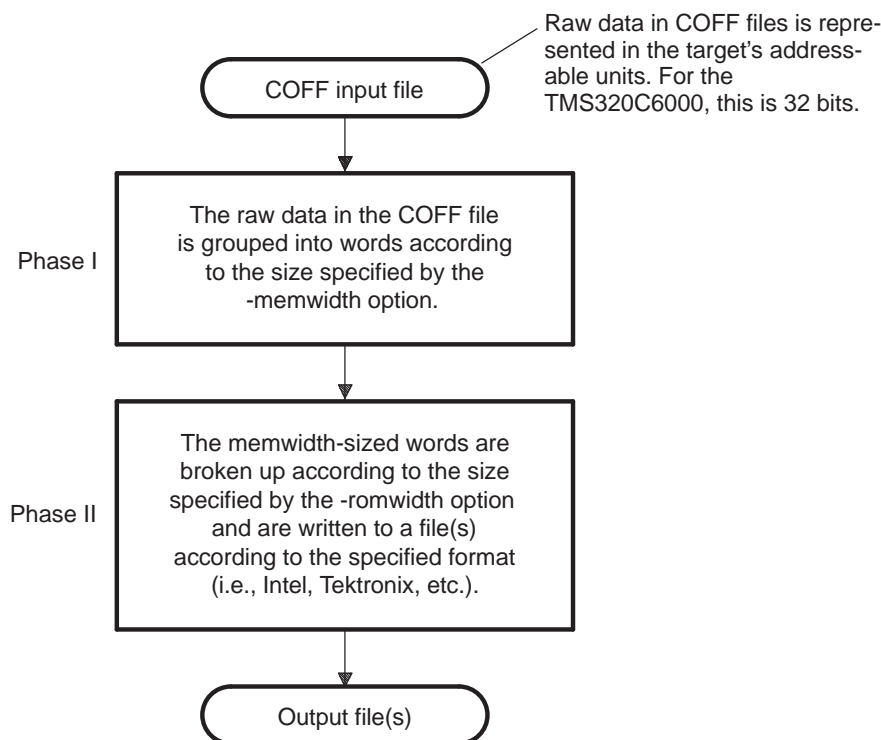
The hex conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex conversion utility, you must understand how the utility treats word widths. Three widths are important in the conversion process:

- Target width
- Memory width
- ROM width

The terms target word, memory word, and ROM word refer to a word of such a width.

Figure 11-2 illustrates the two separate and distinct phases of the hex conversion utility's process flow.

**Figure 11-2. Hex Conversion Utility Process Flow**



#### 11.3.1 Target Width

Target width is the unit size (in bits) of the target processor's word. The unit size corresponds to the data bus size on the target processor. The width is fixed for each target and cannot be changed. The TMS320C6000 targets have a width of 32 bits.

### 11.3.2 Specifying the Memory Width

Memory width is the physical width (in bits) of the memory system. Usually, the memory system is physically the same width as the target processor width: a 32-bit processor has a 32-bit memory architecture. However, some applications require target words to be broken into multiple, consecutive, and narrower memory words.

By default, the hex conversion utility sets memory width to the target width (in this case, 32 bits).

You can change the memory width by:

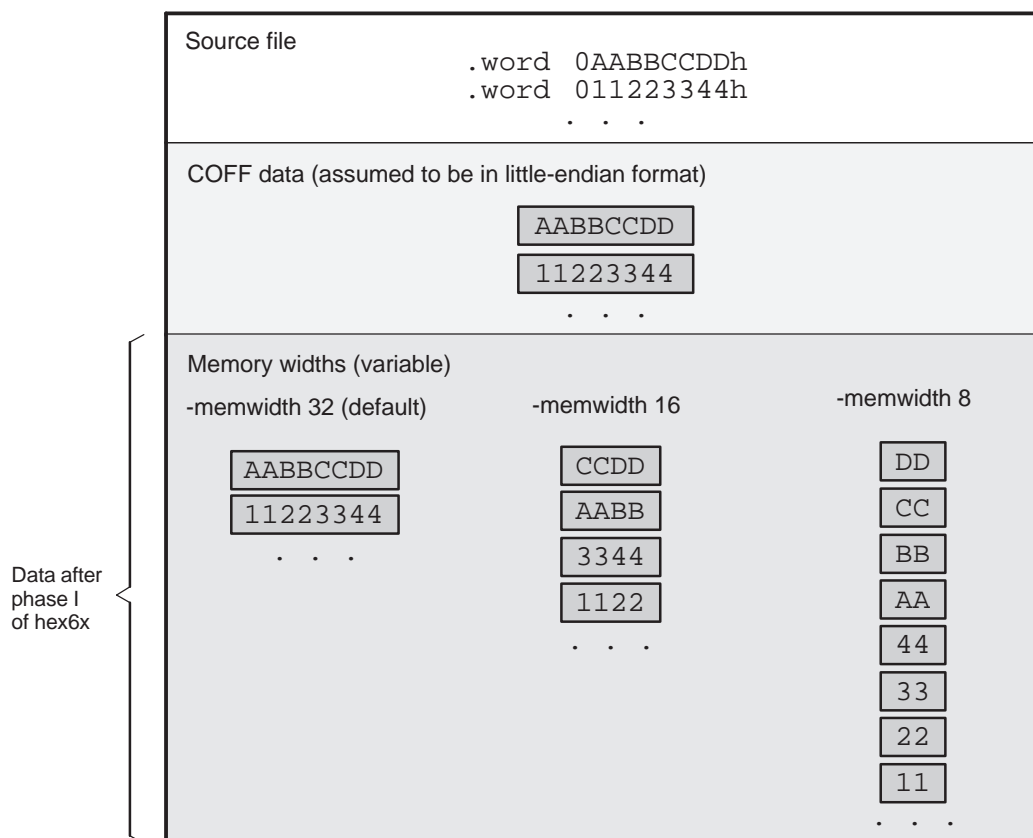
- Using the **-memwidth** option. This changes the memory width value for the entire file.
- Setting the **memwidth** parameter of the ROMS directive. This changes the memory width value for the address range specified in the ROMS directive and overrides the -memwidth option for that range. See [Section 11.4](#).

For both methods, use a value that is a power of 2 greater than or equal to 8.

You should change the memory width default value of 32 only when you need to break single target words into consecutive, narrower memory words.

[Figure 11-3](#) demonstrates how the memory width is related to COFF data.

**Figure 11-3. COFF Data and Memory Widths**



### 11.3.3 Partitioning Data Into Output Files

ROM width specifies the physical width (in bits) of each ROM device and corresponding output file (usually one byte or eight bits). The ROM width determines how the hex conversion utility partitions the data into output files. After the COFF data is mapped to the memory words, the memory words are broken into one or more output files. The number of output files is determined by the following formulas:

- If memory width  $\geq$  ROM width:  
number of files = memory width  $\div$  ROM width
- If memory width  $<$  ROM width:  
number of files = 1

For example, for a memory width of 32, you could specify a ROM width value of 32 and get a single output file containing 32-bit words. Or you can use a ROM width value of 16 to get two files, each containing 16 bits of each word.

The default ROM width that the hex conversion utility uses depends on the output format:

- All hex formats except TI-Tagged are configured as lists of 8-bit bytes; the default ROM width for these formats is 8 bits.
- TI-Tagged is a 16-bit format; the default ROM width for TI-Tagged is 16 bits.

---

#### The TI-Tagged Format is 16 Bits Wide

**Note:** You cannot change the ROM width of the TI-Tagged format. The TI-Tagged format supports a 16-bit ROM width only.

---

You can change ROM width (except for TI-Tagged format) by:

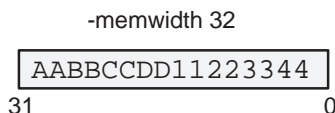
- Using the **-romwidth** option. This option changes the ROM width value for the entire COFF file.
- Setting the **romwidth** parameter of the ROMS directive. This parameter changes the ROM width value for a specific ROM address range and overrides the -romwidth option for that range. See [Section 11.4](#).

For both methods, use a value that is a power of 2 greater than or equal to 8.

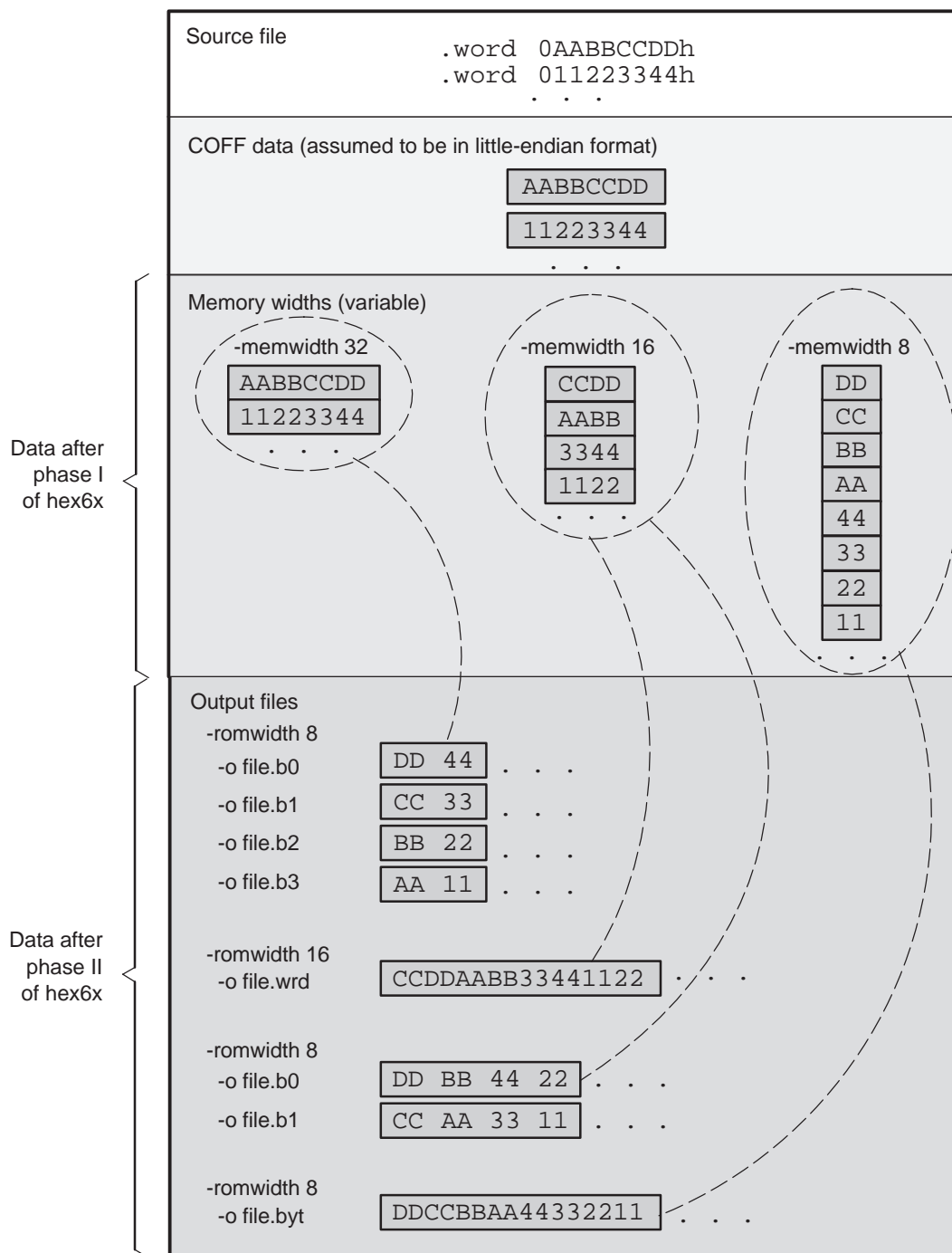
If you select a ROM width that is wider than the natural size of the output format (16 bits for TI-Tagged or 8 bits for all others), the utility simply writes multibyte fields into the file.

[Figure 11-4](#) illustrates how the COFF data, memory, and ROM widths are related to one another.

Memory width and ROM width are used only for grouping the COFF data; they do not represent values. Thus, the byte ordering of the COFF data is maintained throughout the conversion process. To refer to the partitions within a memory word, the bits of the memory word are always numbered from right to left as follows:



**Figure 11-4. Data, Memory, and ROM Widths**



### 11.3.4 Specifying Word Order for Output Words

There are two ways to split a wide word into consecutive memory locations in the same hex conversion utility output file:

- **-order M** specifies **big-endian** ordering, in which the most significant part of the wide word occupies the first of the consecutive locations
- **-order L** specifies **little-endian** ordering, in which the the least significant part of the wide word occupies the first of the consecutive locations

By default, the utility uses little-endian format. Unless your boot loader program expects big-endian format, avoid using -order M.

---

#### When the -order Option Applies

##### Notes:

- This option applies only when you use a memory width with a value of 32 (-memwidth32). Otherwise, the hex utility does not have access to the entire 32-bit word and cannot perform the byte swapping necessary to change the endianness; -order is ignored.
  - This option does not affect the way memory words are split into output files. Think of the files as a set: the set contains a least significant file and a most significant file, but there is no ordering over the set. When you list filenames for a set of files, you always list the least significant first, regardless of the -order option.
- 

## 11.4 The ROMS Directive

The ROMS directive specifies the physical memory configuration of your system as a list of address-range parameters.

Each address range produces one set of files containing the hex conversion utility output data that corresponds to that address range. Each file can be used to program one single ROM device.

The ROMS directive is similar to the MEMORY directive of the TMS320C6000 linker: both define the memory map of the target address space. Each line entry in the ROMS directive defines a specific address range. The general syntax is:

### ROMS

```
{
    romname :    [origin=value,] [length=value,] [romwidth=value,]
                  [ memwidth=value,] [fill=value]
                  [files={filename1, filename2, ...}]

    romname :    [origin=value,] [length=value,] [romwidth=value,]
                  [ memwidth=value,] [fill=value]
                  [files={filename1, filename2, ...}]

    ...
}
```

<b>ROMS</b>	begins the directive definition.
<i>romname</i>	identifies a memory range. The name of the memory range can be one to eight characters in length. The name has no significance to the program; it simply identifies the range. (Duplicate memory range names are allowed.)
<b>origin</b>	specifies the starting address of a memory range. It can be entered as origin, org, or o. The associated value must be a decimal, octal, or hexadecimal constant. If you omit the origin value, the origin defaults to 0. The following table summarizes the notation you can use to specify a decimal, octal, or hexadecimal constant:

Constant	Notation	Example
Hexadecimal	0x prefix or h suffix	0x77 or 077h
Octal	0 prefix	077
Decimal	No prefix or suffix	77

<b>length</b>	specifies the length of a memory range as the physical length of the ROM device. It can be entered as length, len, or l. The value must be a decimal, octal, or hexadecimal constant. If you omit the length value, it defaults to the length of the entire address space.
<b>romwidth</b>	specifies the physical ROM width of the range in bits (see <a href="#">Section 11.3.3</a> ). Any value you specify here overrides the -romwidth option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8.
<b>memwidth</b>	specifies the memory width of the range in bits (see <a href="#">Section 11.3.2</a> ). Any value you specify here overrides the -memwidth option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8. <i>When using the memwidth parameter, you must also specify the paddr parameter for each section in the SECTIONS directive. (See <a href="#">Section 11.5</a>.)</i>
<b>fill</b>	specifies a fill value to use for the range. In image mode, the hex conversion utility uses this value to fill any holes between sections in a range. A hole is an area between the input sections that comprises an output section that contains no actual code or data. The fill value must be a decimal, octal, or hexadecimal constant with a width equal to the target width. Any value you specify here overrides the -fill option. When using fill, you must also use the -image command line option. (See <a href="#">Section 11.8.2</a> .)
<b>files</b>	identifies the names of the output files that correspond to this range. Enclose the list of names in curly braces and order them from <i>least significant</i> to <i>most significant</i> output file, where the bits of the memory word are numbered from right to left. The number of file names must equal the number of output files that the range generates. To calculate the number of output files, see <a href="#">Section 11.3.3</a> . The utility warns you if you list too many or too few filenames.

Unless you are using the -image option, all of the parameters that define a range are optional; the commas and equal signs are also optional. A range with no origin or length defines the entire address space. In image mode, an origin and length are required for all ranges.

Ranges must not overlap and must be listed in order of ascending address.

#### 11.4.1 When to Use the ROMS Directive

If you do not use a ROMS directive, the utility defines a single default range that includes the entire address space. This is equivalent to a ROMS directive with a single range without origin or length.

Use the ROMS directive when you want to:

- **Program large amounts of data into fixed-size ROMs.** When you specify memory ranges corresponding to the length of your ROMs, the utility automatically breaks the output into blocks that fit into the ROMs.
- **Restrict output to certain segments.** You can also use the ROMS directive to restrict the conversion to a certain segment or segments of the target address space. The utility does not convert the data that falls outside of the ranges defined by the ROMS directive. Sections can span range boundaries; the utility splits them at the boundary into multiple ranges. If a section falls completely outside any of the ranges you define, the utility does not convert that section and issues no messages or warnings. Thus, you can exclude sections without listing them by name with the SECTIONS directive. However, if a section falls partially in a range and partially in unconfigured memory, the utility issues a warning and converts only the part within the range.
- **Use image mode.** When you use the -image option, you must use a ROMS directive. Each range is filled completely so that each output file in a range contains data for the whole range. Holes before, between, or after sections are filled with the fill value from the ROMS directive, with the value specified with the -fill option, or with the default value of 0.

### 11.4.2 An Example of the ROMS Directive

The ROMS directive in [Example 11-1](#) shows how 16K bytes of 16-bit memory could be partitioned for two 8K-byte 8-bit EPROMs. [Figure 11-5](#) illustrates the input and output files.

#### Example 11-1. A ROMS Directive Example

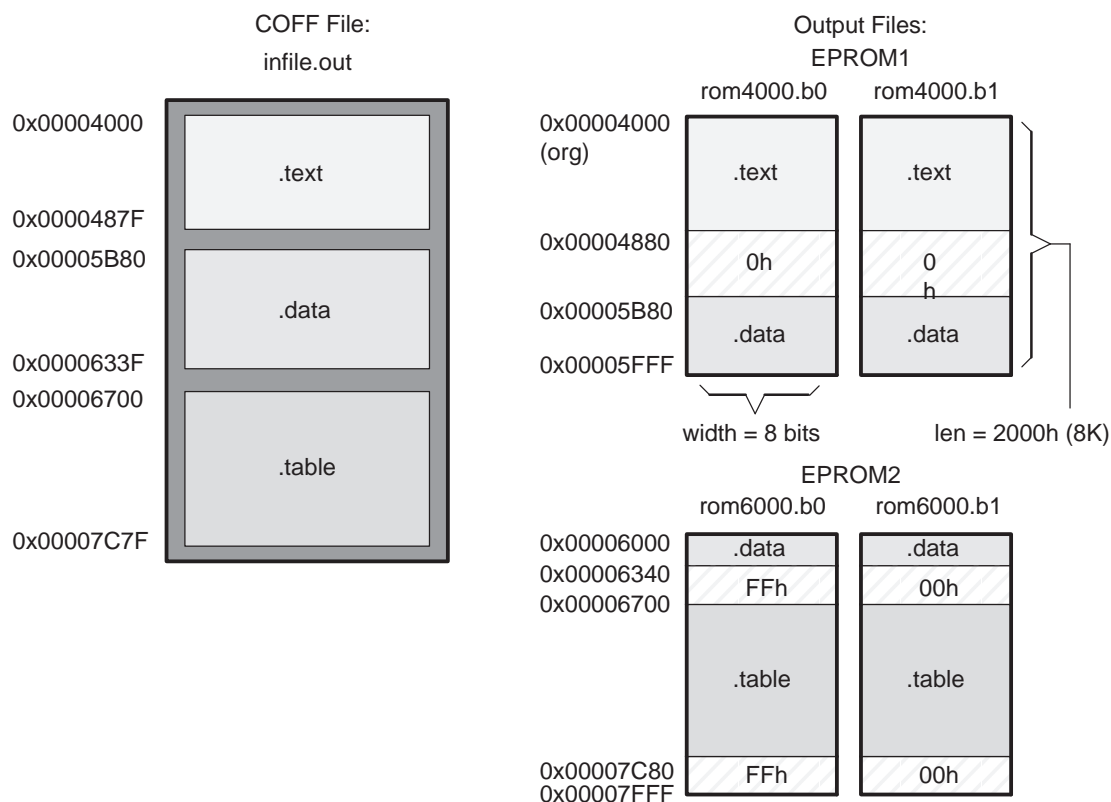
```
infile.out
-image
-memwidth 16

ROMS
{
    EPROM1: org = 0x00004000, len = 0x2000, romwidth = 8
           files = { rom4000.b0, rom4000.b1}

    EPROM2: org = 0x00006000, len = 0x2000, romwidth = 8,
           fill = 0xFF00FF00,
           files = { rom6000.b0, rom6000.b1}
}
```



**Figure 11-5. The infile.out File Partitioned Into Four Output Files**



The map file (specified with the -map option) is advantageous when you use the ROMS directive with multiple ranges. The map file shows each range, its parameters, names of associated output files, and a list of contents (section names and fill values) broken down by address. [Example 11-2](#) is a segment of the map file resulting from the example in [Example 11-1](#).

**Example 11-2. Map File Output From [Example 11-1](#) Showing Memory Ranges**

```
-----
00004000..00005fff Page=0 Width=8 "EPROM1"
-----
OUTPUT FILES:  rom4000.b0  [b0..b7]
                rom4000.b1  [b8..b15]
CONTENTS: 00004000..0000487f .text
           00004880..00005b7f FILL = 00000000
           00005b80..00005fff .data
-----
00006000..00007fff Page=0 Width=8 "EPROM2"
-----
OUTPUT FILES:  rom6000.b0  [b0..b7]
                rom6000.b1  [b8..b15]
CONTENTS: 00006000..0000633f .data
           00006340..000066ff FILL = ff00ff00
           00006700..00007c7f .table
           00007c80..00007fff FILL = ff00ff00
```

### The **SECTIONS** Directive

EPROM1 defines the address range from 0x00004000 through 0x00005FFF with the following sections:

This section ...	Has this range ...
.text	0x00004000 through 0x0000487F
.data	0x00005B80 through 0x00005FFF

The rest of the range is filled with 0h (the default fill value), converted into two output files:

- rom4000.b0 contains bits 0 through 7
- rom4000.b1 contains bits 8 through 15

EPROM2 defines the address range from 0x00006000 through 0x00007FFF with the following sections:

This section ...	Has this range ...
.data	0x00006000 through 0x0000633F
.table	0x00006700 through 0x00007C7F

The rest of the range is filled with 0xFF00FF00 (from the specified fill value). The data from this range is converted into two output files:

- rom6000.b0 contains bits 0 through 7
- rom6000.b1 contains bits 8 through 15

## 11.5 The **SECTIONS** Directive

You can convert specific sections of the COFF file by name with the hex conversion utility **SECTIONS** directive. You can also specify those sections that you want to locate in ROM at a different address than the *load* address specified in the linker command file. If you:

- Use a **SECTIONS** directive, the utility converts only the sections that you list in the directive and ignores all other sections in the COFF file.
- Do not use a **SECTIONS** directive, the utility converts all initialized sections that fall within the configured memory. For the TMS320C6000 these sections are .text, .const, and .cinit.

Uninitialized sections are *never* converted, whether or not you specify them in a **SECTIONS** directive.

---

### Sections Generated by the C/C++ Compiler

**Note:** The TMS320C6000 C/C++ compiler automatically generates these sections:

- **Initialized sections:** .text, .const, .cinit, and .switch
  - **Uninitialized sections:** .bss, .stack, and .system
- 

Use the **SECTIONS** directive in a command file. (For more information, see [Section 11.2.2](#).) The general syntax for the **SECTIONS** directive is:

### **SECTIONS**

```
{
    sname[:] [paddr=value]
    sname[:] [paddr=boot]
    sname[:] [boot]
    ...
}
```

### **SECTIONS**

begins the directive definition.

*sname*

identifies a section in the COFF input file. If you specify a section that does not exist, the utility issues a warning and ignores the name.

<b>paddr=</b> <i>value</i>	specifies the physical ROM address at which this section should be located. This value overrides the section load address given by the linker. This value must be a decimal, octal, or hexadecimal constant. It can also be the word <b>boot</b> (to indicate a boot table section for use with a boot loader). <i>If your file contains multiple sections, and if one section uses a paddr parameter, then all sections must use a paddr parameter.</i>
<b>boot</b>	configures a section for loading by a boot loader. This is equivalent to using <b>paddr=boot</b> . Boot sections have a physical address determined by the location of the boot table. The origin of the boot table is specified with the -bootorg option.

For more similarity with the linker's SECTIONS directive, you can use colons after the section names (in place of the equal sign on the boot keyboard). For example, the following statements are equivalent:

```
SECTIONS { .text: .data: boot }
SECTIONS { .text: .data = boot }
```

In the example below, the COFF file contains six initialized sections: .text, .data, .const, .vectors, .coeff, and .tables. Suppose you want only .text and .data to be converted. Use a SECTIONS directive to specify this:

```
SECTIONS { .text: .data: }
```

To configure both of these sections for boot loading, add the boot keyword:

```
SECTIONS { .text = boot .data = boot }
```

#### Using the -boot Option and the SECTIONS Directive

**Note:** When you use the SECTIONS directive with the boot table (-boot) option, the -boot option is ignored. You must explicitly specify any boot sections in the SECTIONS directive. For more information about -boot and other command line options associated with boot tables, see [Section 11.2](#).

## 11.6 Excluding a Specified Section

The -exclude *section\_name* option can be used to inform the hex utility to ignore the specified section. If a SECTIONS directive is used, it overrides the -exclude option.

For example, if a SECTIONS directive containing the section name *mysect* is used and an -exclude *mysect* is specified, the SECTIONS directive takes precedence and *mysect* is not excluded.

The -exclude option has a limited wildcard capability. The \* character can be placed at the beginning or end of the name specifier to indicate a suffix or prefix, respectively. For example, -exclude sect\* disqualifies all sections that begin with the characters sect.

If you specify the -exclude option on the command line with the \* wildcard, enter quotes around the section name and wildcard. For example, -exclude"sect\*". Using quotes prevents the \* from being interpreted by the hex conversion utility. If -exclude is in a command file, then the quotes should not be specified.

## 11.7 Assigning Output Filenames

When the hex conversion utility translates your COFF object file into a data format, it partitions the data into one or more output files. When multiple files are formed by splitting memory words into ROM words, *filenames are always assigned in order from least to most significant*, where bits in the memory words are numbered from right to left. This is true, regardless of target or COFF endian ordering.

The hex conversion utility follows this sequence when assigning output filenames:

1. **It looks for the ROMS directive.** If a file is associated with a range in the ROMS directive and you have included a list of files (files = { . . . }) on that range, the utility takes the filename from the list. For example, assume that the target data is 32-bit words being converted to four files, each eight bits wide. To name the output files using the ROMS directive, you could specify:

```
ROMS
{
    RANGE1: romwidth=8, files={ xyz.b0 xyz.b1 xyz.b2 xyz.b3 }
}
```

The utility creates the output files by writing the least significant bits to xyz.b0 and the most significant bits to xyz.b3.

2. **It looks for the -o options.** You can specify names for the output files by using the -o option. If no filenames are listed in the ROMS directive and you use -o options, the utility takes the filename from the list of -o options. The following line has the same effect as the example above using the ROMS directive:

```
-o xyz.b0 -o xyz.b1 -o xyz.b2 -o xyz.b3
```

If both the ROMS directive and -o options are used together, the ROMS directive overrides the -o options.

3. **It assigns a default filename.** If you specify no filenames or fewer names than output files, the utility assigns a default filename. A default filename consists of the base name from the COFF input file plus a 2- to 3-character extension. The extension has three parts:

- a. A format character, based on the output format (see [Section 11.11](#) for more information):

<b>a</b>	for ASCII-Hex
<b>i</b>	for Intel
<b>m</b>	for Motorola-S
<b>t</b>	for TI-Tagged
<b>x</b>	for Tektronix

- b. The range number in the ROMS directive. Ranges are numbered starting with 0. If there is no ROMS directive, or only one range, the utility omits this character.

- c. The file number in the set of files for the range, starting with 0 for the least significant file.

For example, assume coff.out is for a 32-bit target processor and you are creating Intel format output. With no output filenames specified, the utility produces four output files named coff.i0, coff.i1, coff.i2, coff.i3.

If you include the following ROMS directive when you invoke the hex conversion utility, you would have eight output files:

```
ROMS
{
    range1: o = 0x00001000 l = 0x1000
    range2: o = 0x00002000 l = 0x1000
}
```

These output files ...	Contain data in these locations ...
coff.i00, coff.i01, coff.i01	0x00001000 through 0x00001FFF
coff.i02, coff.i03	0x00002000 through 0x00002FFF

## 11.8 Image Mode and the -fill Option

This section points out the advantages of operating in image mode and describes how to produce output files with a precise, continuous image of a target memory range.

### 11.8.1 Generating a Memory Image

With the -image option, the utility generates a memory image by completely filling all of the mapped ranges specified in the ROMS directive.

A COFF file consists of blocks of memory (sections) with assigned memory locations. Typically, all sections are not adjacent: there are holes between sections in the address space for which there is no data. When such a file is converted *without* the use of image mode, the hex conversion utility bridges these holes by using the address records in the output file to skip ahead to the start of the next section. In other words, there may be discontinuities in the output file addresses. Some EPROM programmers do not support address discontinuities.

In image mode, there are no discontinuities. Each output file contains a continuous stream of data that corresponds exactly to an address range in target memory. Any holes before, between, or after sections are filled with a fill value that you supply.

An output file converted by using image mode still has address records, because many of the hexadecimal formats require an address on each line. However, in image mode, these addresses are always contiguous.

---

#### Defining the Ranges of Target Memory

**Note:** If you use image mode, you must also use a ROMS directive. In image mode, each output file corresponds directly to a range of target memory. You must define the ranges. If you do not supply the ranges of target memory, the utility tries to build a memory image of the entire target processor address space- potentially a huge amount of output data. To prevent this situation, the utility requires you to explicitly restrict the address space with the ROMS directive.

---

### 11.8.2 Specifying a Fill Value

The -fill option specifies a value for filling the holes between sections. The fill value must be specified as an integer constant following the -fill option. The width of the constant is assumed to be that of a word on the target processor. For example, specifying -fill 0FFFFh results in a fill pattern of 0000FFFFh. The constant value is not sign extended.

The hex conversion utility uses a default fill value of 0 if you do not specify a value with the fill option. *The -fill option is valid only when you use -image; otherwise, it is ignored.*

### 11.8.3 Steps to Follow in Using Image Mode

- Step 1:** Define the ranges of target memory with a ROMS directive. See [Section 11.4](#) for details.
- Step 2:** Invoke the hex conversion utility with the -image option. You can optionally use the -zero option to reset the address origin to 0 for each output file. If you do not specify a fill value with the ROMS directive and you want a value other than the default of 0, use the -fill option.

## 11.9 Building a Table for an On-Chip Boot Loader

On the C621x, C671x, and C64x devices, a ROM boot process is supported where the EDMA copies 1K bytes from the beginning of CE1 (EMIFB CE1 on C64x) to address 0, using default ROM timings. After the transfer, the CPU begins executing from address 0. In this mode a second level boot load typically occurs, due to the limited amount of memory copied at boot.

The hex conversion utility supports the second level boot loader by automatically building the boot table.

### 11.9.1 Description of the Boot Table

The input for a boot loader is the boot table. The boot table contains records that instruct the boot loader to copy blocks of data contained in the table to specified destination addresses. The hex conversion utility automatically builds the boot table for the boot loader. Using the utility, you specify the COFF sections you want the boot loader to initialize through the boot table, the table location, and the name of the section containing the boot loader and where it should be located. The hex conversion utility builds a complete image of the table and converts it into hexadecimal in the output files. Then, you can burn the table into ROM.

### 11.9.2 The Boot Table Format

The boot table format is simple. There is a header record containing a 4 byte field that indicates where the boot loader should branch after it has completed copying data. After the header, each COFF section that is to be included in the boot table will have the following:

1. 4 byte field containing the size of the section
2. 4 byte field containing the destination address for the copy
3. The actual data to be copied

Multiple sections can be entered; a termination block containing a 4 byte field of zeros follows the last COFF section.

Section 1 Size
Section 1 Dest
Section 1 Data
Section 2 Size
Section 2 Dest
Section 2 Data
Section N Size
Section N Dest
Section N Data
0x00000000

### 11.9.3 How to Build the Boot Table

[Table 11-2](#) summarizes the hex conversion utility options available for the boot loader.

**Table 11-2. Boot-Loader Options**

Option	Description
-boot	Convert all sections into bootable form (use instead of a SECTIONS directive).
-bootorg <i>value</i>	Specify the source address of the boot loader table.
-bootsection <i>sectname value</i>	Specify the section name <i>sectname</i> containing the boot loader routine. The <i>value</i> argument tells the hex utility where to place the boot loader routine.
-e <i>value</i>	Specify the entry point at which to begin execution after boot loading. The <i>value</i> can be an address or a global symbol.

### 11.9.3.1 Building the Boot Table

To build the boot table, follow these steps:

- Step 1: Link the file.** Each block of the boot table data corresponds to an initialized section in the COFF file. Uninitialized sections are not converted by the hex conversion utility (see [Section 11.5](#)). You must link into your application a boot loader routine that will read the boot table and perform the copy operations. It should be linked to its eventual run-time address.
- When you select a section for placement in a boot-loader table, the hex conversion utility places the section's *load address* in the destination address field for the block in the boot table. The section content is then treated as raw data for that block. *The hex conversion utility does not use the section run address.* When linking, you need not worry about the ROM address or the construction of the boot table- the hex conversion utility handles this.
- Step 2: Identify the bootable sections.** You can use the -boot option to tell the hex conversion utility to configure all sections for boot loading. Or, you can use a SECTIONS directive to select specific sections to be configured (see [Section 11.5](#)). Note that if you use a SECTIONS directive, the -boot option is ignored.
- Step 3: Set the ROM address of the boot table.** Use the -bootorg option to set the source address of the complete table. For example, if you are using the C6711 and booting from memory location 0x90000400, specify -bootorg 0x90000400. The address field for the boot table in the the hex conversion utility output file will then start at 0x90000400.
- If you do not use the -bootorg option at all, the utility places the table at the origin of the first memory range in a ROMS directive. If you do not use a ROMS directive, the table will start at the first section load address.
- Step 4: Set boot-loader-specific options.** Set entry point. If -e is not used to set the entry point, then it will default to the entry point indicated in the COFF object file.
- Step 5: Describe the boot routine.** If the boot option is used, then you should use the -bootsection option to indicate to the hex utility which COFF section contains the boot routine. This option will prevent the boot routine from being in the boot table. The -bootsection option also indicates to the hex utility where the routine should be placed in ROM. For the C621x, C671x, and C64x devices, this address would typically be the beginning of CE1 (EMIFB CE1 on C64x). This option is ignored if -boot is not used.
- When the SECTIONS directive is used to explicitly identify which sections should exits in the boot table, use the PADDR section option to indicate where the boot routine section will exist.
- Step 6: Describe your system memory configuration.** See [Section 11.3](#), and [Section 11.4](#).

### 11.9.3.2 Leaving Room for the Boot Table

The complete boot table is similar to a single section containing all of the header records and data for the boot loader. The address of this "section" is the boot table origin. As part of the normal conversion process, the hex conversion utility converts the boot table to hexadecimal format and maps it into the output files like any other section.

### Building a Table for an On-Chip Boot Loader

Be sure to leave room in your system memory for the boot table, especially when you are using the ROMS directive. The boot table cannot overlap other nonboot sections or unconfigured memory. Usually, this is not a problem; typically, a portion of memory in your system is reserved for the boot table. Simply configure this memory as one or more ranges in the ROMS directive, and use the `-bootorg` option to specify the starting address.

#### 11.9.3.3 Setting the Entry Point for the Boot Table

After the boot routine finishes copying data, it branches to the entry point defined the COFF object file. By using the `-e` option with the hex conversion utility, you can set the entry point to a different address.

For example, if you want your program to start running at address 0123h after loading, specify `-e 0123h` on the command line or in a command file. You can determine the `-e` address by looking at the map file that the linker generates.

#### Valid Entry Points

**Note:** The value can be a constant, or it can be a symbol that is externally defined (for example, with a `.global`) in the assembly source.

#### 11.9.4 Using the C6x Boot Loader

This subsection explains how to use the hex conversion utility with the boot loader for C6x devices through sample hex utility command files. [Example 11-3](#) uses the `SECTIONS` directive to specify exactly which COFF sections will be placed in the boot table.

##### Example 11-3. Sample Command File for Booting From a C6x EPROM

```

abc.out          /* input file          */
-a              /* ascii format          */
-image          /* create complete ROM image */
-zero          /* reset address origin to 0 */
-memwidth 8     /* 8-bit memory          */
-map abchex.map /* create a hex map file   */
-bootorg 0x90000400 /* external memory boot */

ROMS
{
    FLASH: org=0x90000000, len=0x20000, romwidth=8, files={abc.hex}
}

SECTIONS
{
    .boot_load:  PADDR=0x90000000
    .text:       BOOT
    .cinit:       BOOT
    .const:       BOOT
}

```

[Example 11-4](#) does not explicitly name the boot sections with the `SECTIONS` directive. Instead, it uses the `-boot` option to indicate that all initialized sections should be placed in the boot table. It also uses the `-bootsection` option to distinguish the section containing the boot routine.

##### Example 11-4. Alternative Sample Command File for Booting From a C6x EPROM

```

abc.out          /* input file          */
-a              /* ascii format          */
-image          /* create complete Rom image */
-zero          /* reset address origin to 0 */
-memwidth 8     /* 8-bit memory          */

```



**Example 11-4. Alternative Sample Command File for Booting From a C6x EPROM (continued)**

```
-map abchex.map                /* create a hex map file      */
-boot                        /* create boot table      */
-bootorg 0x90000400          /* external memory boot   */
-bootsection .boot_load 0x90000000 /* give boot section & addr */

ROMS
{
    FLASH: org=0x90000000, len=0x20000, romwidth=8, files={abc.hex}
}
```

## 11.10 Controlling the ROM Device Address

The hex conversion utility output address field corresponds to the ROM device address. The EPROM programmer burns the data into the location specified by the hex conversion utility output file address field. The hex conversion utility offers some mechanisms to control the starting address in ROM of each section. However, many EPROM programmers offer direct control of the location in ROM in which the data is burned.

Depending on whether or not you are using the boot loader, the hex conversion utility output file controlling mechanisms are different.

**Non-boot loader mode.** The address field of the hex conversion utility output file is controlled by the following mechanisms listed from low to high priority:

1. **The linker command file.** By default, the address field of the hex conversion utility output file is the load address (as given in the linker command file).
2. **The paddr parameter of the SECTIONS directive.** When the paddr parameter is specified for a section, the hex conversion utility bypasses the section load address and places the section in the address specified by paddr.
3. **The -zero option.** When you use the -zero option, the utility resets the address origin to 0 for each output file. Since each file starts at 0 and counts upward, any address records represent offsets from the beginning of the file (the address within the ROM) rather than actual target addresses of the data. You must use the -zero option in conjunction with the -image option to force the starting address in each output file to be zero. If you specify the -zero option without the -image option, the utility issues a warning and ignores the -zero option.

**Boot-Loader Mode.** When the boot loader is used, the hex conversion utility places the different COFF sections that are in the boot table into consecutive memory locations. Each COFF section becomes a boot table block whose destination address is equal to the linker-assigned section load address.

In a boot table, the address field of the hex conversion utility output file is not related to the section load addresses assigned by the linker. The address fields of the boot table are simply offsets to the beginning of the table. The section load addresses assigned by the linker will be encoded into the boot table along with the size of the section and the data contained within the section. These addresses will be used to store the data into memory during the boot load process.

The beginning of the boot table defaults to the linked load address of the first bootable section in the COFF input file, unless you use one of the following mechanisms, listed here from low to high priority. Higher priority mechanisms override the values set by low priority options in an overlapping range.

1. **The ROM origin specified in the ROMS directive.** The hex conversion utility places the boot table at the origin of the first memory range in a ROMS directive.
2. **The -bootorg option.** The hex conversion utility places the boot table at the address specified by the -bootorg option if you select boot loading from memory.

## 11.11 Description of the Object Formats

The hex conversion utility has options that identify each format. [Table 11-3](#) specifies the format options. They are described in the following sections.

- You need to use only one of these options on the command line. If you use more than one option, the last one you list overrides the others.
- The default format is Tektronix (-x option).

**Table 11-3. Options for Specifying Hex Conversion Formats**

Option	Format	Address Bits	Default Width
-a	ASCII-Hex	16	8
-i	Intel	32	8
-m	Motorola-S	32	8
-t	TI-Tagged	16	16
-x	Tektronix	32	8

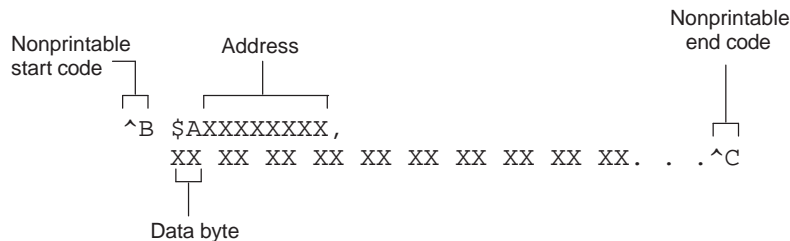
**Address bits** determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

The **default width** determines the default output width of the format. You can change the default width by using the -romwidth option or by using the romwidth parameter in the ROMS directive. You cannot change the default width of the TI-Tagged format, which supports a 16-bit width only.

### 11.11.1 ASCII-Hex Object Format (-a Option)

The ASCII-Hex object format supports 16-bit addresses. The format consists of a byte stream with bytes separated by spaces. [Figure 11-6](#) illustrates the ASCII-Hex format.

**Figure 11-6. ASCII-Hex Object Format**



The file begins with an ASCII STX character (ctrl-B, 02h) and ends with an ASCII ETX character (ctrl-C, 03h). Address records are indicated with \$Axxxxxxx, in which xxxxxxxx is a 8-digit (32-bit) hexadecimal address. The address records are present only in the following situations:

- When discontinuities occur
- When the byte stream does not begin at address 0

You can avoid all discontinuities and any address records by using the -image and -zero options. This creates output that is simply a list of byte values.

### 11.11.2 Intel MCS-86 Object Format (-i Option)

The Intel object format supports 16-bit addresses and 32-bit extended addresses. Intel format consists of a 9-character (4-field) prefix (which defines the start of record, byte count, load address, and record type), the data, and a 2-character checksum suffix.

The 9-character prefix represents three record types:

Record Type	Description
00	Data record
01	End-of-file record
04	Extended linear address record

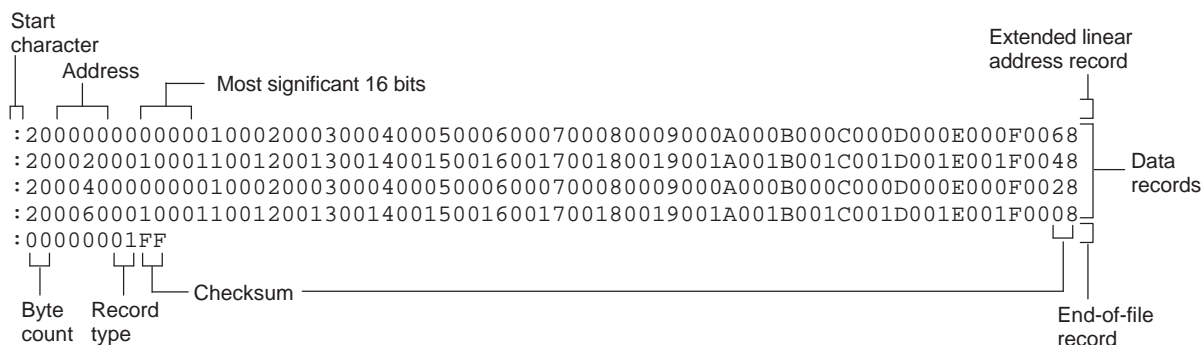
Record type 00, the data record, begins with a colon ( : ) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended linear address) record to create a full 32-bit address. The checksum is the 2s complement (in binary form) of the preceding bytes in the record, including byte count, address, and data bytes.

Record type 01, the end-of-file record, also begins with a colon ( : ), followed by the byte count, the address, the record type (01), and the checksum.

Record type 04, the extended linear address record, specifies the upper 16 address bits. It begins with a colon ( : ), followed by the byte count, a dummy address of 0h, the record type (04), the most significant 16 bits of the address, and the checksum. The subsequent address fields in the data records contain the least significant bytes of the address.

Figure 11-7 illustrates the Intel hexadecimal object format.

Figure 11-7. Intel Hexadecimal Object Format

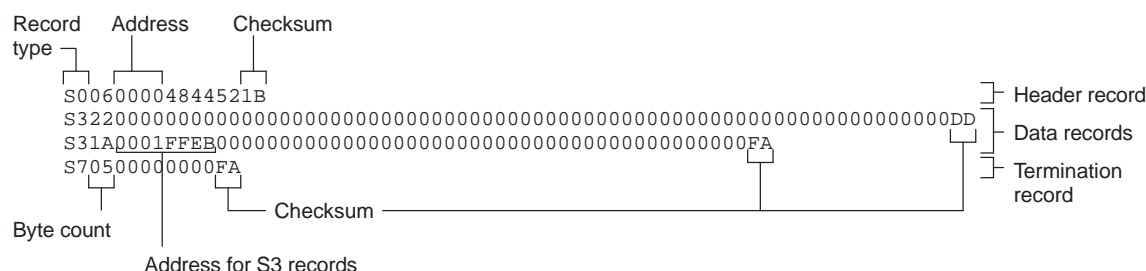


### 11.11.3 Motorola Exorciser Object Format (-m Option)

Record Type	Description
S0	Header record
S3	Code/data record
S7	Termination record

The checksum is the least significant byte of the 1s complement of the sum of the values represented by the pairs of characters making up the byte count, address, and the code/data fields.

### Figure 11-8. Motorola-S Format



The Texas Instruments SDSMAC (TI-Tagged) object format supports 16-bit addresses, including start-of-file record, data records, and end-of-file record. Each data records consists of a series of small fields and is signified by a tag character:

Tag Character	Description
K	Followed by the program identifier
7	Followed by a checksum
8	Followed by a dummy checksum (ignored)
9	Followed by a 16-bit load address
B	Followed by a data word (four characters)
F	Identifies the end of a data record
*	Followed by a data byte (two characters)



## 11.12 Hex Conversion Utility Error Messages

### section mapped to reserved memory message

*Description* A section is mapped into a reserved memory area, listed in the processor memory map.

*Action* Correct the section's allocation or boot-loader address. For valid memory locations, refer to the *TMS320C6200 CPU and Instruction Set Reference Guide*.

### sections overlapping

*Description* Two or more COFF section load addresses overlap or a boot table address overlaps another section.

*Action* This problem may be caused by an incorrect translation (from the load address to the hexadecimal output file address) that is performed by the hex conversion utility when the memory width is less than the data width. See [Section 11.3](#), and [Section 11.10](#).

### unconfigured memory error

*Description* The COFF file contains a section whose load address falls outside the memory range defined in the ROMS directive.

*Action* Correct the ROM range as defined by the ROMS directive to cover the memory range as needed, or modify the section load address. Remember that if the ROMS directive is not used, the memory range defaults to the entire processor address space. For this reason, removing the ROMS directive could also be a workaround.

## ***Sharing C/C++ Header Files With Assembly Source***

---

---

---

You can use the `.cdecls` assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a `.cdecls` block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code.

Topic	Page
12.1 Overview of the <code>.cdecls</code> Directive .....	<b>264</b>
12.2 Notes on C/C++ Conversions.....	<b>264</b>
12.3 Notes on C++ Specific Conversions .....	<b>268</b>
12.4 New Assembler Support .....	<b>269</b>

## 12.1 Overview of the .cdecls Directive

The .cdecls directive allows programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between the C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically. This allows the programmer to reference the C/C++ constructs in assembly code — calling functions, allocating space, and accessing structure members — using the equivalent assembly mechanisms. While function and variable definitions are ignored, most common C/C++ elements are converted to assembly: enumerations, (non function-like) macros, function and variable prototypes, structures, and unions.

See [the .cdecls topic](#) for details on the syntax of the .cdecls assembler directive.

The .cdecls directive can appear anywhere in an assembly source file, and can occur multiple times within a file. However, the C/C++ environment created by one .cdecls is **not** inherited by a later .cdecls; the C/C++ environment starts over for each .cdecls instance.

For example, the following code causes the warning to be issued:

```
.cdecls C,NOLIST
%{
    #define ASMTEST 1
}%

.cdecls C,NOLIST
%{
    #ifndef ASMTEST
        #warn "ASMTEST not defined!" /* will be issued */
    #endif
}%
```

Therefore, a typical use of the .cdecls block is expected to be a single usage near the beginning of the assembly source file, in which all necessary C/C++ header files are included.

Use the compiler `-ipath` (include path) options to specify additional include file paths needed for the header files used in assembly, as you would when compiling C files.

Any C/C++ errors or warnings generated by the code of the .cdecls are emitted as they normally would for the C/C++ source code. C/C++ errors cause the directive to fail, and any resulting converted assembly is not included.

C/C++ constructs that cannot be converted, such as function-like macros or variable definitions, cause a comment to be output to the converted assembly file. For example:

```
; ASM HEADER WARNING - variable definition 'ABCD' ignored
```

The prefix ASM HEADER WARNING appears at the beginning of each message. To see the warnings, either the WARN parameter needs to be specified so the messages are displayed on STDERR, or else the LIST parameter needs to be specified so the warnings appear in the listing file, if any.

Finally, note that the converted assembly code does not appear in the same order as the original C/C++ source code and C/C++ constructs may be simplified to a normalized form during the conversion process, but this should not affect their final usage.

## 12.2 Notes on C/C++ Conversions

The following sections describe C and ++ conversion elements that you need to be aware of when sharing header files with assembly source.

### 12.2.1 Comments

Comments are consumed entirely at the C level, and do not appear in the resulting converted assembly file.



### 12.2.2 Conditional Compilation (#if/#else/#ifdef/etc.)

Conditional compilation is handled entirely at the C level during the conversion step. Define any necessary macros either on the command line (using the compiler -DNAME=value option) or within a .cdecls block using #define. The #if, #ifdef, etc. C/C++ directives are **not** converted to assembly .if, .else, .elseif, and .endif directives.

### 12.2.3 Pragmas

Pragmas found in the C/C++ source code cause a warning to be generated as they are not converted. They have no other effect on the resulting assembly file. See for the WARN and NOWARN parameter discussion for where these warnings are created.

### 12.2.4 The #error and #warning Directives

These preprocessor directives are handled completely by the compiler during the parsing step of conversion. If one of these directives is encountered, the appropriate error or warning message is emitted. These directives are not converted to .emsg or .wmsg in the assembly output.

### 12.2.5 Predefined symbol \_\_ASM\_HEADER\_\_

The C/C++ macro \_\_ASM\_HEADER\_\_ is defined in the compiler while processing code within .cdecls. This allows you to make changes in your code, such as not compiling definitions, during the .cdecls processing.

---

#### Be Careful With the \_\_ASM\_HEADER\_\_ Macro

**Note:** You must be very careful not to use this macro to introduce any changes in the code that could result in inconsistencies between the code processed while compiling the C/C++ source and while converting to assembly.

---

### 12.2.6 Usage Within C/C++ asm( ) Statements

The .cdecls directive is not allowed within C/C++ asm( ) statements and will cause an error to be generated.

### 12.2.7 The #include Directive

The C/C++ #include preprocessor directive is handled transparently by the compiler during the conversion step. Such #includes can be nested as deeply as desired as in C/C++ source. The assembly directives .include and .copy are not used or needed within a .cdecls. Use the command line -lpath option to specify additional paths to be searched for included files, as you would for C compilation.

### 12.2.8 Conversion of #define Macros

Only object-like macros are converted to assembly. Function-like macros have no assembly representation and so cannot be converted. Pre-defined and built-in C/C++ macros are not converted to assembly (i.e., \_\_FILE\_\_, \_\_TIME\_\_, \_\_TI\_COMPILER\_VERSION\_\_, etc.). For example, this code is converted to assembly because it is an object-like macro:

```
#define NAME Charley
```

This code is not converted to assembly because it is a function-like macro:

```
#define MAX(x,y) (x>y ? x : y)
```

Some macros, while they are converted, have no functional use in the containing assembly file. For example, the following results in the assembly substitution symbol FOREVER being set to the value while(1), although this has no useful use in assembly because while(1) is not legal assembly code.

```
#define FOREVER while(1)
```

Macro values are **not** interpreted as they are converted. For example, the following results in the assembler substitution symbol OFFSET being set to the literal string value 5+12 and **not** the value 17. This happens because the semantics of the C/C++ language require that macros are evaluated in context and not when they are parsed.

```
#define OFFSET 5+12
```

Because macros in C/C++ are evaluated in their usage context, C/C++ printf escape sequences such as `\n` are not converted to a single character in the converted assembly macro. See [Section 12.2.11](#) for suggestions on how to use C/C++ macro strings.

Macros are converted using the new `.define` directive (see [Section 12.4.2](#)), which functions similarly to the `.asg` assembler directive. The exception is that `.define` disallows redefinitions of register symbols and mnemonics to prevent the conversion from corrupting the basic assembly environment. To remove a macro from the assembly scope, `.undef` can be used following the `.cdecls` that defines it (see [Section 12.4.3](#)).

The macro functionality of `#` (stringize operator) is only useful within functional macros. Since functional macros are not supported by this process, `#` is not supported either. The concatenation operator `##` is only useful in a functional context, but can be used degenerately to concatenate two strings and so it is supported in that context.

### 12.2.9 The `#undef` Directive

Symbols undefined using the `#undef` directive before the end of the `.cdecls` are not converted to assembly.

### 12.2.10 Enumerations

Enumeration members are converted to `.enum` elements in assembly. For example:

```
enum state { ACTIVE=0x10, SLEEPING=0x01, INTERRUPT=0x100, POWEROFF, LAST};
```

is converted to the following assembly code:

```
state      .enum
ACTIVE     .emember 16
SLEEPING   .emember 1
INTERRUPT   .emember 256
POWEROFF   .emember 257
LAST       .emember 258
           .endenum
```

The members are used via the pseudo-scoping created by the `.enum` directive:

```
AC0 = #(state.ACTIVE)
```

The usage is similar to that for accessing structure members, `enum_name.member`.

This pseudo-scoping is used to prevent enumeration member names from corrupting other symbols within the assembly environment.

### 12.2.11 C Strings

Because C string escapes such as `\n` and `\t` are not converted to hex characters `0x0A` and `0x09` until their use in a string constant in a C/C++ program, C macros whose values are strings cannot be represented as expected in assembly substitution symbols. For example:

```
#define MSG "\tHI\n"
```

becomes, in assembly:

```
.define "" "\tHI\n",MSG ; 6 quoted characters! not 5!
```

When used in a C string context, you expect this statement to be converted to 5 characters (tab, H, I, newline, NULL), but the `.string` assembler directive does not know how to perform the C escape conversions.

You can use the new `.cstring` directive to cause the escape sequences and NULL termination to be properly handled as they would in C/C++. Using the above symbol `MSG` with a `.cstring` directive results in 5 characters of memory being allocated, the same characters as would result if used in a C/C++ strong context. (See [Section 12.4.7](#) for the `.cstring` directive syntax.)

### 12.2.12 C/C++ Built-In Functions

The C/C++ built-in functions, such as `sizeof( )`, are not translated to their assembly counterparts, if any, if they are used in macros. Also, their C expression values are not inserted into the resulting assembly macro because macros are evaluated in context and there is no active context when converting the macros to assembly.

Suitable functions such as `$sizeof( )` are available in assembly expressions. However, as the basic types such as `int/char/float` have no type representation in assembly, there is no way to ask for `$sizeof(int)`, for example, in assembly.

### 12.2.13 Structures and Unions

C/C++ structures and unions are converted to assembly `.struct` and `.union` elements. Padding and ending alignments are added as necessary to make the resulting assembly structure have the same size and member offsets as the C/C++ source. The primary purpose is to allow access to members of C/C++ structures, as well as to facilitate debugging of the assembly code. For nested structures, the assembly `.tag` feature is used to refer to other structures/unions.

The alignment is also passed from the C/C++ source so that the assembly symbol is marked with the same alignment as the C/C++ symbol. (See [Section 12.2.3](#) for information about pragmas, which may attempt to modify structures.) Because the alignment of structures is stored in the assembly symbol, built-in assembly functions like `$sizeof( )` and `$alignof( )` can be used on the resulting structure name symbol.

When using unnamed structures (or unions) in typedefs, such as:

```
typedef struct { int a_member; } mystrname;
```

This is really a shorthand way of writing:

```
struct temporary_name { int a_member; };
typedef temporary_name mystrname;
```

The conversion processes the above statements in the same manner: generating a temporary name for the structure and then using `.define` to output a typedef from the temporary name to the user name. You should use your `mystrname` in assembly the same as you would in C/C++, but do not be confused by the assembly structure definition in the list, which contains the temporary name. You can avoid the temporary name by specifying a name for the structure, as in:

```
typedef struct a_st_name { ... } mystrname;
```

If a shorthand method is used in C to declare a variable with a particular structure, for example:

```
extern struct a_name { int a_member; } a_variable;
```

Then after the structure is converted to assembly, a `.tag` directive is generated to declare the structure of the external variable, such as:

```
_a_variable .tag a_st_name
```

This allows you to refer to `_a_variable.a_member` in your assembly code.

### 12.2.14 Function/Variable Prototypes

Non-static function and variable prototypes (not definitions) will result in a `.global` directive being generated for each symbol found.

See [Section 12.3.1](#) for C++ name mangling issues.

Function and variable definitions will result in a warning message being generated (see the `WARN/NOWARN` parameter discussion for where these warnings are created) for each, and they will not be represented in the converted assembly.

## Notes on C++ Specific Conversions

---

The assembly symbol representing the variable declarations will not contain type information about those symbols. Only a .global will be issued for them. Therefore, it is your responsibility to ensure the symbol is used appropriately.

See [Section 12.2.13](#) for information on variables names which are of a structure/union type.

### 12.2.15 C Constant Suffixes

The C constant suffixes u, l, and f are passed to the assembly unchanged. The assembler will ignore these suffixes if used in assembly expressions.

### 12.2.16 Basic C/C++ Types

Only complex types (structures and unions) in the C/C++ source code are converted to assembly. Basic types such as int, char, or float are not converted or represented in assembly beyond any existing .int, .char, .float, etc. directives that previously existed in assembly.

Typedefs of basic types are therefore also not represented in the converted assembly.

## 12.3 Notes on C++ Specific Conversions

The following sections describe C++ specific conversion elements that you need to be aware of when sharing header files with assembly source.

### 12.3.1 Name Mangling

Symbol names may be mangled in C++ source files. When mangling occurs, the converted assembly will use the mangled names to avoid symbol name clashes. You can use the demangler (dem430) to demangle names and identify the correct symbols to use in assembly.

To defeat name mangling in C++ for symbols where polymorphism (calling a function of the same name with different kinds of arguments) is not required, use the following syntax:

```
extern "C" void somefunc(int arg);
```

The above format is the short method for declaring a single function. To use this method for multiple functions, you can also use the following syntax:

```
extern "C"
{
    void somefunc(int arg);
    int  anotherfunc(int arg);
    ...
}
```

### 12.3.2 Derived Classes

Derived classes are only partially supported when converting to assembly because of issues related to C++ scoping which does not exist in assembly. The greatest difference is that base class members do not automatically become full (top-level) members of the derived class. For example:

```
-----
class base
{
    public:
        int b1;
};

class derived : public base
{
    public:
        int d1;
}
-----
```

In C++ code, the class derived would contain both integers b1 and d1. In the converted assembly structure "derived", the members of the base class must be accessed using the name of the base class, such as `derived.__b_base.b1` rather than the expected `derived.b1`.

A non-virtual, non-empty base class will have `__b_` prepended to its name within the derived class to signify it is a base class name. That is why the example above is `derived.__b_base.b1` and not simply `derived.base.b1`.

### 12.3.3 Templates

No support exists for templates.

### 12.3.4 Virtual Functions

No support exists for virtual functions, as they have no assembly representation.

## 12.4 New Assembler Support

### 12.4.1 Enumerations (*.enum/.emember/.endenum*)

New directives have been created to support a pseudo-scoping for enumerations.

The format of these new directives is:

```
ENUM_NAME    .enum
MEMBER1      .emember [value]
MEMBER2      .emember [value]
...
                .endenum
```

The **.enum** directive begins the enumeration definition and **.endenum** terminates it.

The enumeration name (*ENUM\_NAME*) cannot be used to allocate space; its size is reported as zero.

The format to use the value of a member is *ENUM\_NAME.MEMBER*, similar to a structure member usage.

The **.emember** directive optionally accepts the value to set the member to, just as in C/C++. If not specified, the member takes a value one more than the previous member. As in C/C++, member names cannot be duplicated, although values can be. Unless specified with **.emember**, the first enumeration member will be given the value 0 (zero), as in C/C++.

The **.endenum** directive cannot be used with a label, as structure **.endstruct** directives can, because the **.endenum** directive has no value like the **.endstruct** does (containing the size of the structure).

Conditional compilation directives (**.if/.else/.elseif/.endif**) are the only other non-enumeration code allowed within the **.enum/.endenum** sequence.

### 12.4.2 The *.define* Directive

The new **.define** directive functions in the same manner as the existing **.asg** directive, except that **.define** disallows creation of a substitution symbol that has the same name as a register symbol or mnemonic. It does not create a new symbol name space in the assembler, rather it uses the existing substitution symbol name space. The syntax for the directive is:

```
.define substitution string, substitution symbol name
```

The **.define** directive is used to prevent corruption of the assembly environment when converting C/C++ headers.

### 12.4.3 The **.undef/.unasg** Directives

The **.undef** directive is used to remove the definition of a substitution symbol created using **.define** or **.asg**. This directive will remove the named symbol from the substitution symbol table from the point of the **.undef** to the end of the assembly file. The syntax for these directives is:

**.undef** *substitution symbol name*

**.unasg** *substitution symbol name*

This can be used to remove from the assembly environment any C/C++ macros that may cause a problem.

Also see [Section 12.4.2](#), which covers the **.define** directive.

### 12.4.4 The **\$defined( )** Directive

The **\$defined** directive returns true/1 or false/0 depending on whether the name exists in the current substitution symbol table or the standard symbol table. In essence **\$defined** returns TRUE if the assembler has any user symbol in scope by that name. This differs from **\$isdefed** in that **\$isdefed** only tests for NON-substitution symbols. The syntax is:

**\$defined(** *substitution symbol name***)**

A statement such as **".if \$defined(macroname)"** is then similar to the C code **"#ifdef macroname"**.

See [Section 12.4.2](#) and [Section 12.4.3](#) for the use of **.define** and **.undef** in assembly.

### 12.4.5 The **\$sizeof Built-In Function**

The new assembly built-in function **\$sizeof( )** can be used to query the size of a structure in assembly. It is an alias for the already existing **\$structsz( )**. The syntax is:

**\$sizeof(** *structure name***)**

The **\$sizeof** function can then be used similarly to the C built-in function **sizeof( )**.

The assembler's **\$sizeof( )** built-in function cannot be used to ask for the size of basic C/C++ types, such as **\$sizeof(int)**, because those basic type names are not represented in assembly. Only complex types are converted from C/C++ to assembly.

Also see [Section 12.2.12](#), which notes that this conversion does not happen automatically if the C/C++ **sizeof( )** built-in function is used within a macro.

### 12.4.6 Structure/Union Alignment & **\$alignof( )**

The assembly **.struct** and **.union** directives now take an optional second argument which can be used to specify a minimum alignment to be applied to the symbol name. This is used by the conversion process to pass the specific alignment from C/C++ to assembly.

The assembly built-in function **\$alignof( )** can be used to report the alignment of these structures. This can be used even on assembly structures, and the function will return the minimum alignment calculated by the assembler.

### 12.4.7 The **.cstring** Directive

You can use the new **.cstring** directive to cause the escape sequences and NULL termination to be properly handled as they would in C/C++.

**.cstring** "String with C escapes.\nWill be NULL terminated.\012"

See [Section 12.2.11](#) for more information on the new **.cstring** directive.

## Common Object File Format

The assembler and linker 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.

*Sections* are a basic COFF concept. [Chapter 2](#) discusses COFF sections in detail. If you understand section operation, you can use the assembly language tools more efficiently.

This appendix contains technical details about the TMS320C6000™ 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 appendix is to provide supplementary information on the internal format of COFF object files.

Topic	Page
<b>A.1 COFF File Structure .....</b>	<b><a href="#">272</a></b>
<b>A.2 File Header Structure.....</b>	<b><a href="#">273</a></b>
<b>A.3 Optional File Header Format .....</b>	<b><a href="#">274</a></b>
<b>A.4 Section Header Structure .....</b>	<b><a href="#">274</a></b>
<b>A.5 Structuring Relocation Information .....</b>	<b><a href="#">276</a></b>
<b>A.6 Symbol Table Structure and Content .....</b>	<b><a href="#">277</a></b>

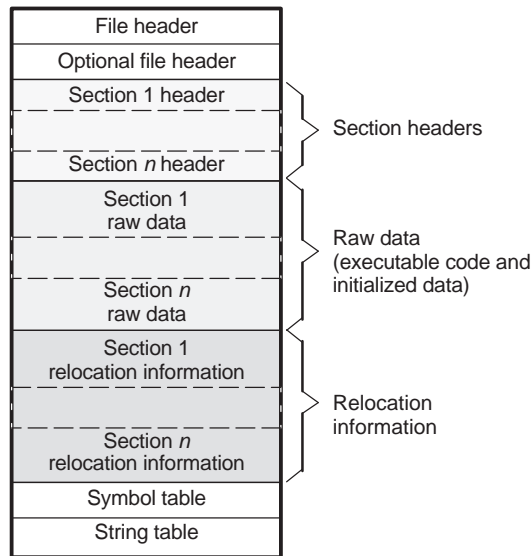
## A.1 COFF File Structure

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 linker 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 A-1](#) illustrates the object file structure.

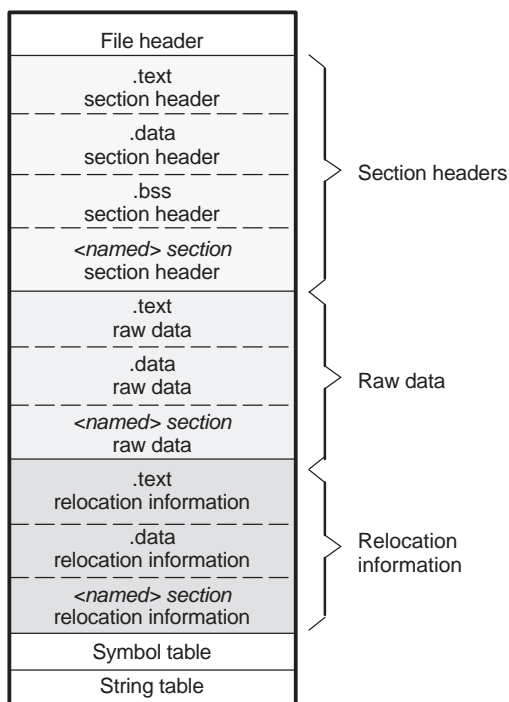
**Figure A-1. COFF File Structure**



[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 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 A-2. Sample COFF Object File**



## A.2 File Header Structure

The file header contains 22 bytes of information that describe the general format of an object file. [Table A-1](#) shows the structure of the C6000 COFF file header.

**Table A-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 <a href="#">Table A-2</a> )
20-21	Unsigned short	Target ID; magic number (0099h) indicates the file can be executed in a C6000 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, both F\_RELFLG and F\_EXEC are set).

**Table A-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)
	0004h	Reserved
F_LSYMS	0008h	Local symbols were stripped from the file
F_LITTLE	0100h	The target is a little-endian device
F_BIG	0200h	The target is a big-endian device

### 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	Optional file header magic number (0108h)
2-3	Short	Version stamp
4-7	Integer	Size (in bytes) of executable code
8-11	Integer	Size (in bytes) of initialized data
12-15	Integer	Size (in bytes) of uninitialized data
16-19	Integer	Entry point
20-23	Integer	Beginning address of executable code
24-27	Integer	Beginning address of initialized data

### A.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 A-4](#) shows the structure of each section header.

**Table A-4. 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	Integer	Section's physical address
12-15	Integer	Section's virtual address
16-19	Integer	Section size in bytes
20-23	Integer	File pointer to raw data
24-27	Integer	File pointer to relocation entries
28-31	Integer	Reserved
32-35	Unsigned integer	Number of relocation entries
36-39	Unsigned integer	Reserved
40-43	Unsigned integer	Flags (see <a href="#">Table A-5</a> )
44-45	Unsigned short	Reserved
46-47	Unsigned short	Memory page number

[Table A-5](#) lists the flags that can appear in bytes 36 through 39 of the section header.

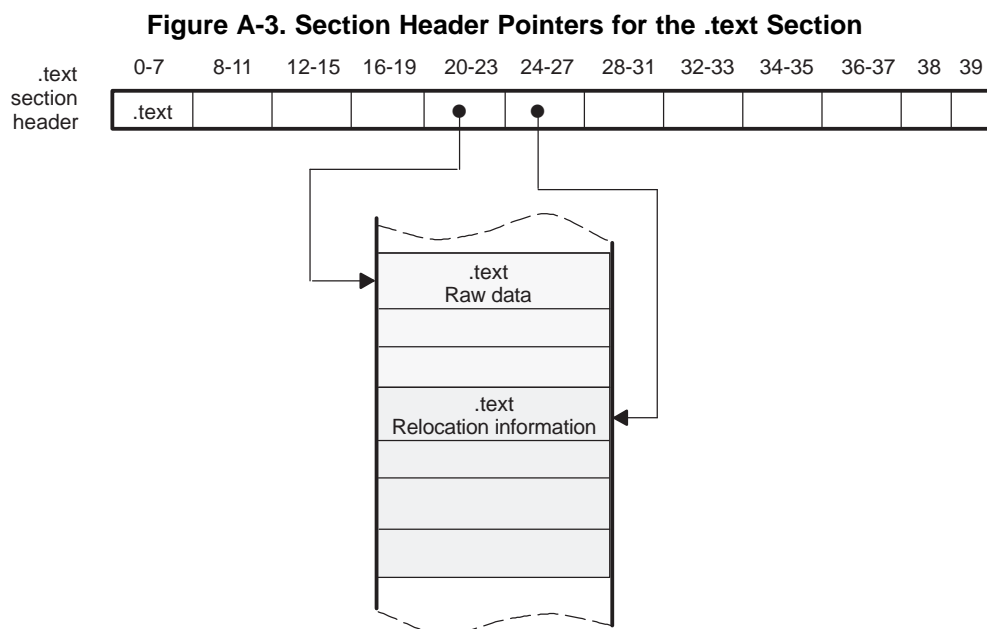
**Table A-5. Section Header Flags (Bytes 40 Through 43)**

Mnemonic	Flag	Description
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_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	00001000h	Alignment used as a blocking factor
STYP_PASS	00002000h	Section should pass through unchanged
STYP_CLINK	00004000h	Section requires conditional linking
STYP_VECTOR	00008000h	Section contains vector table
STYP_PADDED	00010000h	Section has been padded

The flags listed in [Table A-5](#) 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  $2^{(\text{value of bits 8-11})}$ . For example if bits 8-11 are 0101b (decimal integer 5), then the alignment is 32 ( $2^5$ ).

[Figure A-3](#) illustrates how the pointers in a section header point to the elements in an object file that are associated with the .text section.



As [Figure A-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. 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 each input section are treated.

COFF file relocation information entries use the 10-byte format shown in [Table A-6](#).

**Table A-6. Relocation Entry Contents**

Byte Number	Type	Description
0-3	Integer	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 <a href="#">Table A-7</a> )

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:

```

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, 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. [Table A-7](#) lists the relocation types.

**Table A-7. 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_C60LO16	0054h	MVK instruction low half register
R_C60HI16	0055h	MVKH or MVKLH high half register
R_C60SECT	0056h	Section-based offset
R_C60PCR10	0053h	10-bit Packet PC Relative (BDEC, BPOS)

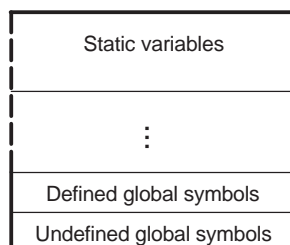
**Table A-7. Relocation Types (Bytes 8 and 9) (continued)**

Mnemonic	Flag	Relocation Type
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)
RE_ADD	4000h	Operator instruction +
RE_SUB	4001h	Operator instruction -
RE_NEG	4002h	Operator instruction unary -
RE_MPY	4003h	Operator instruction *
RE_DIV	4004h	Operator instruction /
RE_MOD	4005h	Operator instruction %
RE_SR	4006h	Unsigned shift right
RE_ASR	4007h	Signed shift right
RE_SL	4008h	Shift left
RE_AND	4009h	AND function
RE_OR	400Ah	OR function
RE_XOR	400Bh	Exclusive OR function
RE_NOTB	400Ch	~
RE_ULDFLD	400Dh	Unsigned relocation field load
RE_SLDFLD	400Eh	Signed relocation field load
RE_USTFLD	400Fh	Unsigned relocation field store
RE_SSTFLD	4010h	Signed relocation field store
RE_XSTFLD	4016h	Signed state is not relevant
RE_PUSH	4011h	Push symbol on the stack
RE_PUSHSV	c011h	Push symbol: SEGVALUE flag is set
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 tos and push copy

## A.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 A-4](#).

**Figure A-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)

## Symbol Table Structure and Content

- Type
- Value
- Section it was defined in
- Storage class

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-8](#). Each symbol may also have an 18-byte auxiliary entry; the special symbols listed in [Table A-9](#) 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-8. 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	Integer	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)

### A.6.1 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 as well as an auxiliary entry. [Table A-9](#) lists these symbols.

**Table A-9. 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

### A.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.

### A.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 A-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 A-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'		

#### A.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 A-10 lists valid storage classes.

**Table A-10. 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	19	Tentative external definition
C_LABEL	6	Label	C_STATLAB	20	Static load time label
C_ULABEL	7	Undefined label	C_EXTLAB	21	External load time label
C_MOS	8	Reserved	C_VARARG	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

#### A.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 linker relocates a section, the value of a relocatable symbol changes accordingly.

### A.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 A-11](#) lists these numbers and the indicated sections.

**Table A-11. Section Numbers**

Mnemonic	Section Number	Description
None	-2	Reserved
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
None	1	.text section (typical)
None	2	.data section (typical)
None	3	.bss section (typical)
None	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, 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.

### A.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 A-12](#) illustrates the format of auxiliary table entries.

**Table A-12. Section Format for Auxiliary Table Entries**

Byte Number	Type	Description
0-3	Integer	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)



## ***Symbolic Debugging Directives***

The assembler supports several directives that the TMS320C6000 C/C++ compiler uses for symbolic debugging. These directives differ for the two debugging formats, DWARF and COFF.

These directives are not meant for use by assembly-language programmers. They require arguments that can be difficult to calculate manually, and their usage must conform to a predetermined agreement between the compiler, the assembler, and the debugger. This appendix documents these directives for informational purposes only.

<b>Topic</b>	<b>Page</b>
<b>B.1 DWARF Debugging Format .....</b>	<b>282</b>
<b>B.2 COFF Debugging Format .....</b>	<b>282</b>
<b>B.3 Debug Directive Syntax .....</b>	<b>283</b>

## B.1 DWARF Debugging Format

A subset of the DWARF symbolic debugging directives are always listed in the assembly language file that the compiler creates for program analysis purposes. To list the complete set used for full symbolic debug, invoke the compiler with the `-g` option, as shown below:

```
cl6x -g -k input_file
```

The `-k` option instructs the compiler to retain the generated assembly file.

To disable the generation of all symbolic debug directives, invoke the compiler with the `-symdebug:none` option:

```
cl6x --symdebug:none -k input_file
```

The DWARF debugging format consists of the following directives:

- The **.dwtag** and **.dwendtag** directives define a Debug Information Entry (DIE) in the `.debug_info` section.
- The **.dwattr** directive adds an attribute to an existing DIE.
- The **.dwpsn** directive identifies the source position of a C/C++ statement.
- The **.dwcie** and **.dwentry** directives define a Common Information Entry (CIE) in the `.debug_frame` section.
- The **.dwfde** and **.dwentry** directives define a Frame Description Entry (FDE) in the `.debug_frame` section.
- The **.dwcfa** directive defines a call frame instruction for a CIE or FDE.

## B.2 COFF Debugging Format

COFF symbolic debug is now obsolete. These directives are supported for backwards-compatibility only. The decision to switch to DWARF as the symbolic debug format was made to overcome many limitations of COFF symbolic debug, including the absence of C++ support.

The COFF debugging format consists of the following directives:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the variable or function.
- The **.stag**, **.etag**, and **.utag** directives define structures, enumerations, and unions, respectively. The **.member** directive specifies a member of a structure, enumeration, or union. The **.eos** directive ends a structure, enumeration, or union definition.
- The **.func** and **.endfunc** directives specify the beginning and ending lines of a C/C++ function.
- The **.block** and **.endblock** directives specify the bounds of C/C++ blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source filename.
- The **.line** directive identifies the line number of a C/C++ source statement.

## B.3 Debug Directive Syntax

Table B-1 is an alphabetical listing of the symbolic debugging directives. For information on the C/C++ compiler, refer to the *TMS320C6000 Optimizing Compiler User's Guide*.

**Table B-1. Symbolic Debugging Directives**

Label	Directive	Arguments
	<b>.block</b>	[beginning line number]
	<b>.dwattr</b>	DIE label,DIE attribute name(DIE attribute value)[,DIE attribute name(attribute value) [, ...]
	<b>.dwcfa</b>	call frame instruction opcode[,operand[,operand]]
CIE label	<b>.dwcie</b>	version , return address register
	<b>.dwentry</b>	
	<b>.dwendtag</b>	
	<b>.dwfde</b>	CIE label
	<b>.dwpsn</b>	" filename ", line number , column number
DIE label	<b>.dwtag</b>	DIE tag name,DIE attribute name(DIE attribute value)[,DIE attribute name(attribute value) [, ...]
	<b>.endblock</b>	[ending line number]
	<b>.endfunc</b>	[ending line number[,register mask[, frame size]]]
	<b>.eos</b>	
	<b>.etag</b>	name[, size]
	<b>.file</b>	" filename "
	<b>.func</b>	[beginning line number]
	<b>.line</b>	line number[, address]
	<b>.member</b>	name, value[, type, storage class, size, tag, dims]
	<b>.stag</b>	name[, size]
	<b>.sym</b>	name, value[, type, storage class, size, tag, dims]
	<b>.utag</b>	name[, size]



## ***XML Link Information File Description***

---

---

---

The linker supports the generation of an XML link information file via the `--xml_link_info file` option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker-generated map file.

As the linker evolves, the XML link information file may be extended to include additional information that could be useful for static analysis of linker results.

This appendix enumerates all of the elements that are generated by the linker into the XML link information file.

Topic	Page
<b>C.1 XML Information File Element Types .....</b>	<b>286</b>
<b>C.2 Document Elements .....</b>	<b>286</b>

## C.1 XML Information File Element Types

These element types will be generated by the linker:

- **Container elements** represent an object that contains other elements that describe the object. Container elements have an id attribute that makes them accessible from other elements.
- **String elements** contain a string representation of their value.
- **Constant elements** contain a 32-bit unsigned long representation of their value (with a 0x prefix).
- **Reference elements** are empty elements that contain an idref attribute that specifies a link to another container element.

In [Section C.2](#), the element type is specified for each element in parentheses following the element description. For instance, the <link\_time> element lists the time of the link execution (string).

## C.2 Document Elements

The root element, or the document element, is **<link\_info>**. All other elements contained in the XML link information file are children of the <link\_info> element. The following sections describe the elements that an XML information file can contain.

### C.2.1 Header Elements

The first elements in the XML link information file provide general information about the linker and the link session:

- The **<banner>** element lists the name of the executable and the version information (string).
- The **<copyright>** element lists the TI copyright information (string).
- The **<link\_time>** is a timestamp representation of the link time (unsigned 32-bit int).
- The **<output\_file>** element lists the name of the linked output file generated (string).
- The **<entry\_point>** element specifies the program entry point, as determined by the linker (container) with two entries:
  - The **<name>** is the entry point symbol name, if any (string).
  - The **<address>** is the entry point address (constant).

#### **Example C-1. Header Element for the hi.out Output File**

```
<banner>TMS320Cxx COFF Linker          Version x.xx (Jan 6 2004)</banner>
<copyright>Copyright (c) 1996-2004 Texas Instruments Incorporated</copyright>
<link_time>0x43dfd8a4</link_time>
<output_file>hi.out</output_file>
<entry_point>
  <name>_c_int00</name>
  <address>0xaf80</address>
</entry_point>
```

### C.2.2 Input File List

The next section of the XML link information file is the input file list, which is delimited with a **<input\_file\_list>** container element. The **<input\_file\_list>** can contain any number of **<input\_file>** elements.

Each **<input\_file>** instance specifies the input file involved in the link. Each **<input\_file>** has an **id** attribute that can be referenced by other elements, such as an **<object\_component>**. An **<input\_file>** is a container element enclosing the following elements:

- The **<path>** element names a directory path, if applicable (string).
- The **<kind>** element specifies a file type, either archive or object (string).
- The **<file>** element specifies an archive name or filename (string).
- The **<name>** element specifies an object file name, or archive member name (string).

#### Example C-2. Input File List for the hi.out Output File

```
<input_file_list>
  <input_file id="f1-1">
    <kind>object</kind>
    <file>hi.obj</file>
    <name>hi.obj</name>
  </input_file>
  <input_file id="f1-2">
    <path>/tools/lib/</path>
    <kind>archive</kind>
    <file>rtsxxx.lib</file>
    <name>boot.obj</name>
  </input_file>
  <input_file id="f1-3">
    <path>/tools/lib/</path>
    <kind>archive</kind>
    <file>rtsxxx.lib</file>
    <name>exit.obj</name>
  </input_file>
  <input_file id="f1-4">
    <path>/tools/lib/</path>
    <kind>archive</kind>
    <file>rtsxxx.lib</file>
    <name>printf.obj</name>
  </input_file>
  ...
</input_file_list>
```

### C.2.3 Object Component List

The next section of the XML link information file contains a specification of all of the object components that are involved in the link. An example of an object component is an input section. In general, an object component is the smallest piece of object that can be manipulated by the linker.

The **<object\_component\_list>** is a container element enclosing any number of **<object\_component>** elements.

Each **<object\_component>** specifies a single object component. Each **<object\_component>** has an **id** attribute so that it can be referenced directly from other elements, such as a **<logical\_group>**. An **<object\_component>** is a container element enclosing the following elements:

- The **<name>** element names the object component (string).
- The **<load\_address>** element specifies the load-time address of the object component (constant).
- The **<run\_address>** element specifies the run-time address of the object component (constant).
- The **<size>** element specifies the size of the object component (constant).
- The **<input\_file\_ref>** element specifies the source file where the object component originated (reference).

#### Example C-3. Object Component List for the fl-4 Input File

```
<object_component id="oc-20">
  <name>.text</name>
  <load_address>0xac00</load_address>
  <run_address>0xac00</run_address>
  <size>0xc0</size>
  <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
  <name>.data</name>
  <load_address>0x80000000</load_address>
  <run_address>0x80000000</run_address>
  <size>0x0</size>
  <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
  <name>.bss</name>
  <load_address>0x80000000</load_address>
  <run_address>0x80000000</run_address>
  <size>0x0</size>
  <input_file_ref idref="fl-4"/>
</object_component>
```



### C.2.4 Logical Group List

The **<logical\_group\_list>** section of the XML link information file is similar to the output section listing in a linker generated map file. However, the XML link information file contains a specification of GROUP and UNION output sections, which are not represented in a map file. There are three kinds of list items that can occur in a **<logical\_group\_list>**:

- The **<logical\_group>** is the specification of a section or GROUP that contains a list of object components or logical group members. Each **<logical\_group>** element is given an id so that it may be referenced from other elements. Each **<logical\_group>** is a container element enclosing the following elements:
  - The **<name>** element names the logical group (string).
  - The **<load\_address>** element specifies the load-time address of the logical group (constant).
  - The **<run\_address>** element specifies the run-time address of the logical group (constant).
  - The **<size>** element specifies the size of the logical group (constant).
  - The **<contents>** element lists elements contained in this logical group (container). These elements refer to each of the member objects contained in this logical group:
    - The **<object\_component\_ref>** is an object component that is contained in this logical group (reference).
    - The **<logical\_group\_ref>** is a logical group that is contained in this logical group (reference).
- The **<overlay>** is a special kind of logical group that represents a UNION, or a set of objects that share the same memory space (container). Each **<overlay>** element is given an id so that it may be referenced from other elements (like from an **<allocated\_space>** element in the placement map). Each **<overlay>** contains the following elements:
  - The **<name>** element names the overlay (string).
  - The **<run\_address>** element specifies the run-time address of overlay (constant).
  - The **<size>** element specifies the size of logical group (constant).
  - The **<contents>** container element lists elements contained in this overlay. These elements refer to each of the member objects contained in this logical group:
    - The **<object\_component\_ref>** is an object component that is contained in this logical group (reference).
    - The **<logical\_group\_ref>** is a logical group that is contained in this logical group (reference).
- The **<split\_section>** is another special kind of logical group that represents a collection of logical groups that is split among multiple memory areas. Each **<split\_section>** element is given an id so that it may be referenced from other elements. The id consists of the following elements.
  - The **<name>** element names the split section (string).
  - The **<contents>** container element lists elements contained in this split section. The **<logical\_group\_ref>** elements refer to each of the member objects contained in this split section, and each element referenced is a logical group that is contained in this split section (reference).

**Example C-4. Logical Group List for the fl-4 Input File**

```

<logical_group_list>
...
<logical_group id="lg-7">
<name>.text</name>
<load_address>0x20</load_address>
<run_address>0x20</run_address>
<size>0xb240</size>
<contents>
<object_component_ref idref="oc-34"/>
<object_component_ref idref="oc-108"/>
<object_component_ref idref="oc-e2"/>
...
</contents>
</logical_group>
...
<overlay id="lg-b">
<name>UNION_1</name>
<run_address>0xb600</run_address>
<size>0xc0</size>
<contents>
<object_component_ref idref="oc-45"/>
<logical_group_ref idref="lg-8"/>
</contents>
</overlay>
...
<split_section id="lg"12">
<name>.task_scn</name>
<size>0x120</size>
<contents>
<logical_group_ref idref="lg"10"/>
<logical_group_ref idref="lg"11"/>
</contents>
...
</logical_group_list>

```

### C.2.5 Placement Map

The **<placement\_map>** element describes the memory placement details of all named memory areas in the application, including unused spaces between logical groups that have been placed in a particular memory area.

The **<memory\_area>** is a description of the placement details within a named memory area (container). The description consists of these items:

- The **<name>** names the memory area (string).
- The **<page\_id>** gives the id of the memory page in which this memory area is defined (constant).
- The **<origin>** specifies the beginning address of the memory area (constant).
- The **<length>** specifies the length of the memory area (constant).
- The **<used\_space>** specifies the amount of allocated space in this area (constant).
- The **<unused\_space>** specifies the amount of available space in this area (constant).
- The **<attributes>** lists the RWXI attributes that are associated with this area, if any (string).
- The **<fill\_value>** specifies the fill value that is to be placed in unused space, if the fill directive is specified with the memory area (constant).
- The **<usage\_details>** lists details of each allocated or available fragment in this memory area. If the fragment is allocated to a logical group, then a **<logical\_group\_ref>** element is provided to facilitate access to the details of that logical group. All fragment specifications include **<start\_address>** and **<size>** elements.
  - The **<allocated\_space>** element provides details of an allocated fragment within this memory area (container):

#### Example C-5. Placement Map for the fl-4 Input File

```
<placement_map>
<memory_area>
<name>PMEM</name>
<page_id>0x0</page_id>
<origin>0x20</origin>
<length>0x100000</length>
<used_space>0xb240</used_space>
<unused_space>0xf4dc0</unused_space>
<attributes>RWXI</attributes>
<usage_details>
<allocated_space>
<start_address>0x20</start_address>
<size>0xb240</size>
<logical_group_ref idref="lg-7"/>
</allocated_space>
<available_space>
<start_address>0xb260</start_address>
<size>0xf4dc0</size>
</available_space>
</usage_details>
</memory_area>
...
</placement_map>
```

### C.2.6 Far Call Trampoline List

The **<far\_call\_trampoline\_list>** is a list of **<far\_call\_trampoline>** elements. The C6000 linker supports the generation of far call trampolines to help a call site reach a destination that is out of range. A far call trampoline function is guaranteed to reach the called function (callee) as it may utilize an indirect call to the called function.

The **<far\_call\_trampoline\_list>** enumerates all of the far call trampolines that are generated by the linker for a particular link. The **<far\_call\_trampoline\_list>** can contain any number of **<far\_call\_trampoline>** elements. Each **<far\_call\_trampoline>** is a container enclosing the following elements:

- The **<callee\_name>** element names the destination function (string).
- The **<callee\_address>** is the address of the called function (constant).
- The **<trampoline\_object\_component\_ref>** is a reference to an object component that contains the definition of the trampoline function (reference).
- The **<trampoline\_address>** is the address of the trampoline function (constant).
- The **<caller\_list>** enumerates all call sites that utilize this trampoline to reach the called function (container).
- The **<trampoline\_call\_site>** provides the details of a trampoline call site (container) and consists of these items:
  - The **<caller\_address>** specifies the call site address (constant).
  - The **<caller\_object\_component\_ref>** is the object component where the call site resides (reference).

#### Example C-6. Fall Call Trampoline List for the fl-4 Input File

```

<far_call_trampoline_list>
...
<far_call_trampoline>
<callee_name>_foo</callee_name>
<callee_address>0x08000030</callee_address>
<trampoline_object_component_ref idref="oc-123"/>
<trampoline_address>0x2020</trampoline_address>
<caller_list>
<call_site>
<caller_address>0x1800</caller_address>
<caller_object_component_ref idref="oc-23"/>
</call_site>
<call_site>
<caller_address>0x1810</caller_address>
<caller_object_component_ref idref="oc-23"/>
</call_site>
</caller_list>
</far_call_trampoline>
...
</far_call_trampoline_list>

```

### C.2.7 Symbol Table

The **<symbol\_table>** contains a list of all of the global symbols that are included in the link. The list provides information about a symbol's name and value. In the future, the symbol\_table list may provide type information, the object component in which the symbol is defined, storage class, etc.

The **<symbol>** is a container element that specifies the name and value of a symbol with these elements:

- The **<name>** element specifies the symbol name (string).
- The **<value>** element specifies the symbol value (constant).

#### **Example C-7. Symbol Table for the fl-4 Input File**

```
<symbol_table>
<symbol>
  <name>_c_int00</name>
  <value>0xaf80</value>
</symbol>
<symbol>
  <name>_main</name>
  <value>0xb1e0</value>
</symbol>
<symbol>
  <name>_printf</name>
  <value>0xac00</value>
</symbol>
...
</symbol_table>
```



## Glossary

---

**absolute address** — An address that is permanently assigned to a TMS320C6000 memory location.

**alignment** — A process in which the linker places an output section at an address that falls on an  $n$ -byte boundary, where  $n$  is a power of 2. You can specify alignment with the SECTIONS linker directive.

**allocation** — A process in which the linker calculates the final memory addresses of output sections.

**American Standard Code for Information Interchange (ASCII)** — A standard computer code for representing and exchanging alphanumeric information.

**archive library** — A collection of individual files that have been grouped into a single file.

**archiver** — A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

**assembler** — A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro directives. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

**assembly-time constant** — A symbol that is assigned a constant value with the .set directive.

**assignment statement** — A statement that assigns a value to a variable.

**autoinitialization** — The process of initializing global C variables (contained in the .cinit section) before beginning program execution.

**auxiliary entry** — The extra entry that a symbol may have in the symbol table and that contains additional information about the symbol (whether it is a filename, a section name, a function name, etc.).

**binding** — A process in which you specify a distinct address for an output section or a symbol.

**big endian** — An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

**block** — A set of declarations and statements that are grouped together with braces.

**.bss** — One of the default COFF sections. You can use the .bss directive to reserve a specified amount of space in the memory map that can later be used for storing data. The .bss section is uninitialized.

**byte** — A sequence of eight adjacent bits operated upon as a unit.

**C/C++ compiler** — A program that translates C/C++ source statements into assembly language source statements.

**command file** — A file that contains options, filenames, directives, or commands for the linker or hex conversion utility.

- comment** — A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.
- common object file format (COFF)** — A binary object file format configured by a standard developed by AT&T. All COFF sections are independently relocatable in memory space; you can place any section into any allocated block of target memory.
- conditional processing** — A method of processing one block of source code or an alternate block of source code, according to the evaluation of a specified expression.
- configured memory** — Memory that the linker has specified for allocation.
- constant** — A numeric value that does not change and that can be used as an operand.
- cross-reference listing** — An output file created by the assembler and appended to the end of the listing file. The cross reference information lists the symbols that were defined, what line they were defined on, which lines referenced them, and the values as determined by the input assembly source file.
- .data** — One of the default COFF sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.
- directives** — Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).
- emulator** — A hardware development system that emulates TMS320C6200 operation.
- entry point** — The starting execution point in target memory.
- executable module** — An object file that has been linked and can be executed in a TMS320C6000 system.
- expression** — A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.
- external symbol** — A symbol that is used in the current program module but is defined in a different program module.
- field** — For the TMS320C6000, a software-configurable data type whose length can be programmed to be any value in the range of 1-32 bits.
- file header** — A portion of a COFF object file that contains general information about the object file, such as the number of section headers, the type of system the object file can be downloaded to, the number of symbols in the symbol table, and the symbol table's starting address.
- global symbol** — A kind of symbol that is either 1) defined in the current module and accessed in another, or 2) accessed in the current module but defined in another.
- GROUP** — An option of the SECTIONS directive that forces specified output sections to be allocated contiguously (as a group).
- hex conversion utility** — A program that accepts COFF files and converts them into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer.
- high-level language debugging** — The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.
- hole** — An area containing no actual code or data. This area is between the input sections that compose an output section.
- incremental linking** — Linking files in several passes. Incremental linking is useful for large applications, because you can partition the application, link the parts separately, and then link all of the parts together.



- initialized section** — A COFF section that contains executable code or initialized data. An initialized section can be built up with the `.data`, `.text`, or `.sect` directive.
- input section** — A section from an object file that will be linked into an executable module.
- label** — A symbol that begins in column 1 of a source statement and corresponds to the address of that statement.
- line-number entry** — An entry in a COFF output module that maps lines of assembly code back to the original C source file that created them.
- linker** — A software tool that combines object files to form an object module that can be allocated into TMS320C6000 system memory and executed by the device.
- listing file** — An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the SPC.
- little endian** — An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*.
- loader** — A device that loads an executable module into TMS320C6000 system memory.
- macro** — A user-defined routine that can be used as an instruction.
- macro call** — The process of invoking a macro.
- macro definition** — A block of source statements that define the name and the code that make up a macro.
- macro library** — An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of `.asm`.
- macro expansion** — The source statements that are substituted for the macro call and are subsequently assembled.
- magic number** — A COFF file header entry that identifies an object file as a module that can be executed by the TMS320C6000.
- map file** — An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which they were defined.
- member** — The elements or variables of a structure, union, archive, or enumeration.
- memory map** — A map of target system memory space that is partitioned into functional blocks.
- mnemonic** — An instruction name that the assembler translates into machine code.
- model statement** — Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.
- named section** — An initialized section that is defined with a `.sect` directive.
- object file** — A file that has been assembled or linked and contains machine-language object code.
- object library** — An archive library made up of individual object files.
- operands** — The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.
- optional header** — A portion of a COFF object file that the linker uses to perform relocation at download time.
- options** — Command parameters that allow you to request additional or specific functions when you invoke a software tool.

- output module** — A linked, executable object file that can be downloaded and executed on a target system.
- output section** — A final, allocated section in a linked, executable module.
- partial linking** — Linking files in several passes. Incremental linking is useful for large applications because you can partition the application, link the parts separately, and then link all of the parts together.
- quiet run** — An option that suppresses the normal banner and the progress information.
- raw data** — Executable code or initialized data in an output section.
- relocation** — A process in which the linker adjusts all the references to a symbol when the symbol's address changes.
- run address** — The address where a section runs.
- section** — A relocatable block of code or data that will ultimately occupy contiguous space in the TMS320C6000 memory map.
- section header** — A portion of a COFF object file that contains information about a section in the file. Each section has its own header; the header points to the section's starting address, contains the section's size, etc.
- section program counter (SPC)** — An element that keeps track of the current location within a section; each section has its own SPC.
- sign extend** — To fill the unused MSBs of a value with the value's sign bit.
- simulator** — A software development system that simulates TMS320C6000 operation.
- source file** — A file that contains C code or assembly language code that will be compiled or assembled to form an object file.
- static variable** — An element whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; the previous value is resumed when the function or program is reentered.
- storage class** — Any entry in the symbol table that indicates how a symbol is accessed.
- string table** — A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead, they are stored in the string table). The name portion of the symbol's entry points to the location of the string in the string table.
- structure** — A collection of one or more variables grouped together under a single name.
- subsection** — A relocatable block of code or data that will ultimately occupy continuous space in the TMS320C6000 memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.
- symbol** — A string of alphanumeric characters that represents an address or a value.
- symbol table** — A portion of a COFF object file that contains information about the symbols that are defined and used by the file.
- symbolic debugging** — The ability of a software tool to retain symbolic information so that it can be used by a debugging tool, such as a simulator or an emulator.
- tag** — An optional *type* name that can be assigned to a structure, union, or enumeration.
- target memory** — Physical memory in a TMS320C6000 system into which executable object code is loaded.

**.text** — One of the default COFF sections. The .text section is an initialized section that contains executable code. You can use the .text directive to assemble code into the .text section.

**unconfigured memory** — Memory that is not defined as part of the memory map and cannot be loaded with code or data.

**uninitialized section** — A COFF section that reserves space in the memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

**UNION** — An option of the SECTIONS directive that causes the linker to allocate the same address to multiple sections.

**union** — A variable that can hold objects of different types and sizes.

**unsigned value** — An element that is treated as a positive number, regardless of its actual sign.

**well-defined expression** — A term or group of terms that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

**word** — A 16-bit addressable location in target memory.

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

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

<b>Products</b>		<b>Applications</b>	
Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>	Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>	Automotive	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>	Broadband	<a href="http://www.ti.com/broadband">www.ti.com/broadband</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>	Digital Control	<a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>	Military	<a href="http://www.ti.com/military">www.ti.com/military</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>	Optical Networking	<a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>	Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Low Power Wireless	<a href="http://www.ti.com/lpw">www.ti.com/lpw</a>	Telephony	<a href="http://www.ti.com/telephony">www.ti.com/telephony</a>
		Video & Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>
		Wireless	<a href="http://www.ti.com/wireless">www.ti.com/wireless</a>

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

Copyright © 2006, Texas Instruments Incorporated