

TMS470R1x Assembly Language Tools User's Guide

Literature Number: SPNU118E
July 2004



Printed on Recycled Paper

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:

Products

Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DSP	dsp.ti.com
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com

Applications

Audio	www.ti.com/audio
Automotive	www.ti.com/automotive
Broadband	www.ti.com/broadband
Digital Control	www.ti.com/digitalcontrol
Military	www.ti.com/military
Optical Networking	www.ti.com/opticalnetwork
Security	www.ti.com/security
Telephony	www.ti.com/telephony
Video & Imaging	www.ti.com/video
Wireless	www.ti.com/wireless

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

Copyright © 2004, Texas Instruments Incorporated

Read This First

About This Manual

The *TMS470R1x Assembly Language Tools User's Guide* tells you how to use these assembly language tools:

- ☐ Assembler
- ☐ Archiver
- ☐ Linker
- ☐ Absolute lister
- ☐ Cross-reference lister
- ☐ Hex conversion utility

How to Use This Manual

The goal of this book is to help you learn how to use the Texas Instruments assembly language tools specifically designed for the TMS470R1x. This book is divided into four parts:

- ☐ **Introductory information**, consisting of Chapters 1 and 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 TMS470R1x tools more efficiently. Read Chapter 2, *Introduction to Common Object File Format*, before using the assembler and linker.
- ☐ **Assembler description**, consisting of Chapters 3 through 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, assembler directives, and macros.

- ❑ **Additional assembly language tools**, consisting of Chapters 6 through 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 Appendixes A through E, provides supplementary information. This portion contains technical data about the internal format and structure of COFF object files. It discusses symbolic debugging directives that the TMS470R1x C compiler uses. Finally, it includes hex conversion utility examples and a glossary.

Notational Conventions

This document uses the following conventions:

- ❑ The TMS470R1x device is referred to as the TMS470.
- ❑ The TMS470 16-bit instruction set is referred to as 16-BIS.
- ❑ The TMS470 32-bit instruction set is referred to as 32-BIS.
- ❑ Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **bold version** of the special typeface for emphasis; 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 2F      x      .byte  47
2 00000001 32      z      .byte  50
3 00000002                .text
```

- ❑ 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. Syntax that is entered on a command line is centered. Syntax that is used in a text file is left-justified. Here is an example of command-line syntax:

abs470 *filename*

abs470 is a command. The command invokes the absolute lister and has one parameter, indicated by *filename*. When you invoke the absolute lister, you supply the name of the file that the absolute lister uses as input.

- 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. This is an example of an instruction syntax with square brackets:

hex470 [*options*] *filename*

The **hex470** 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.

- Braces ({ and }) indicate a list. The symbol | (read as *or*) separates items within the list. This is an example of a list in an instruction syntax:

STM[*cond*]{**FD**|**ED**|**FA**|**EA**|**IA**|**IB**|**DA**|**DB**} *Rn*[!], { *Rlist* }[^]

The list provides eight choices: FD, ED, FA, EA, IA, IB, DA, or DB

You must choose one item from the list; you cannot choose more than one item. Unless the braces are in a **bold** typeface, do not enter the braces when you enter the item you have chosen. (In this example, you enter the braces with *Rlist* because those braces are bold.)

- 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. The syntax for this directive is:

.byte *value*₁ [, ... , *value*_{*n*}]

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.

- 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
PC	Program counter register
0x (prefix)	Hexadecimal integer
Q, q (suffix)	Octal integer
R0–R15	TMS470R1x registers 0 through 15

Related Documentation From Texas Instruments

The following books describe the TMS470R1x 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.

TMS470R1x Optimizing C/C++ Compiler User’s Guide (literature number SPNU151) describes the TMS470R1x C/C++ compiler. This C/C++ compiler accepts ANSI standard C/C++ source code and produces assembly language source code for the TMS470R1x devices.

Code Composer User’s Guide (literature number SPRU328) explains how to use the Code Composer development environment to build and debug embedded real-time DSP applications.

TMS470R1x User’s Guide (literature number SPNU134) describes the TMS470R1x RISC microcontroller, its architecture (including registers), the ICEBreaker module, interfaces (memory, coprocessor, and debugger), 16-bit and 32-bit instruction sets, and electrical specifications.

Trademarks

Intel and MCS-86 are trademarks of Intel Corporation.

Motorola-S is a trademark of Motorola, Inc.

Windows and Windows NT are registered trademarks of Microsoft Corp.

Tektronix is a trademark of Tektronix, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments Incorporated. Trademarks of Texas Instruments include: TI, XDS, Code Composer, and Code Composer Studio.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Contents

1	Introduction to the Software Development Tools	1-1
	<i>Provides an overview of the software development tools.</i>	
1.1	Software Development Tools Overview	1-2
1.2	Tools Descriptions	1-3
2	Introduction to Common Object File Format	2-1
	<i>Common object file format, or COFF, is the object file format used by the TMS470R1x tools. This chapter discusses the basic COFF concept of sections and how they can help you use the assembler and linker more efficiently. Read this chapter before using the assembler and linker.</i>	
2.1	Sections	2-2
2.2	How the Assembler Handles Sections	2-4
2.2.1	Uninitialized Sections	2-4
2.2.2	Initialized Sections	2-6
2.2.3	Named Sections	2-6
2.2.4	Subsections	2-7
2.2.5	Section Program Counters (SPCs)	2-8
2.2.6	An Example That Uses Sections Directives	2-8
2.3	How the Linker Handles Sections	2-11
2.3.1	Default Memory Allocation	2-12
2.3.2	Placing Sections in the Memory Map	2-13
2.4	Relocation	2-13
2.5	Run-Time Relocation	2-15
2.6	Loading a Program	2-16
2.7	Symbols in a COFF File	2-17
2.7.1	External Symbols	2-17
2.7.2	The Symbol Table	2-18

3	Assembler Description	3-1
	<i>Explains how to invoke the assembler and discusses source statement format, valid constants and expressions, and assembler output.</i>	
3.1	Assembler Overview	3-2
3.2	The Assembler's Role in the Software Development Flow	3-3
3.3	Invoking the Assembler	3-4
3.4	Naming Alternate Directories for Assembler Input	3-7
3.4.1	Using the -I Assembler Option	3-7
3.4.2	Using the A_DIR Environment Variable	3-8
3.5	Source Statement Format	3-10
3.5.1	Label Field	3-10
3.5.2	Mnemonic Field	3-11
3.5.3	Operand Field	3-12
3.5.4	Comment Field	3-15
3.6	Constants	3-16
3.6.1	Binary Integers	3-16
3.6.2	Octal Integers	3-16
3.6.3	Decimal Integers	3-16
3.6.4	Hexadecimal Integers	3-17
3.6.5	Character Constants	3-17
3.6.6	Assembly-Time Constants	3-17
3.7	Character Strings	3-18
3.8	Symbols	3-19
3.8.1	Labels	3-19
3.8.2	Local Labels	3-19
3.8.3	Symbolic Constants	3-21
3.8.4	Defining Symbolic Constants (-d Option)	3-21
3.8.5	Predefined Symbolic Constants	3-23
3.8.6	Substitution Symbols	3-25
3.9	Expressions	3-26
3.9.1	Operators	3-27
3.9.2	Expression Overflow and Underflow	3-27
3.9.3	Well-Defined Expressions	3-28
3.9.4	Conditional Expressions	3-28
3.9.5	Relocatable Symbols and Legal Expressions	3-28
3.10	Source Listings	3-31
3.11	Debugging Assembly Source	3-35
3.12	Cross-Reference Listings	3-37

4	Assembler Directives	4-1
	<i>Describes the directives according to function and presents the directives in alphabetical order.</i>	
4.1	Assembler Directives Summary	4-2
4.2	Directives That Define Sections	4-7
4.3	Directives That Change the Instruction Type	4-10
4.4	Directives That Initialize Constants	4-11
4.5	Directive That Aligns the Section Program Counter	4-15
4.6	Directives That Format the Output Listing	4-17
4.7	Directives That Reference Other Files	4-19
4.8	Directives That Enable Conditional Assembly	4-20
4.9	Directives That Define Symbols at Assembly Time	4-21
4.10	Miscellaneous Directives	4-23
4.11	Directives Reference	4-24
5	Macro Language	5-1
	<i>Describes macro directives, substitution symbols used as macro parameters, and how to create macros.</i>	
5.1	Using Macros	5-2
5.2	Defining Macros	5-3
5.3	Macro Parameters/Substitution Symbols	5-5
5.3.1	Directives That Define Substitution Symbols	5-6
5.3.2	Built-In Substitution Symbol Functions	5-7
5.3.3	Recursive Substitution Symbols	5-9
5.3.4	Forced Substitution	5-9
5.3.5	Accessing Individual Characters of Subscripted Substitution Symbols	5-11
5.3.6	Substitution Symbols as Local Variables in Macros	5-12
5.4	Macro Libraries	5-13
5.5	Using Conditional Assembly in Macros	5-14
5.6	Using Labels in Macros	5-16
5.7	Producing Messages in Macros	5-17
5.8	Using Directives to Format the Output Listing	5-19
5.9	Using Recursive and Nested Macros	5-21
5.10	Macro Directives Summary	5-23
6	Archiver Description	6-1
	<i>Describes instructions for invoking the archiver, creating new archive libraries, and modifying existing libraries.</i>	
6.1	Archiver Overview	6-2
6.2	The Archiver's Role in the Software Development Flow	6-3
6.3	Invoking the Archiver	6-4
6.4	Archiver Examples	6-6

7	Linker Description	7-1
	<i>Explains how to invoke the linker, provides details about linker operation, discusses linker directives, and presents a detailed linking example.</i>	
7.1	Linker Overview	7-2
7.2	The Linker's Role in the Software Development Flow	7-3
7.3	Invoking the Linker	7-4
7.4	Linker Options	7-5
7.4.1	Relocation Capabilities (<code>-a</code> and <code>-r</code> Options)	7-6
7.4.2	Create an Absolute Listing File (<code>-abs</code> Option)	7-7
7.4.3	Allocate Memory for Use by the Loader to Pass Arguments (<code>--args</code> Option)	7-8
7.4.4	Disable Merge of Symbolic Debugging Information (<code>-b</code> Option)	7-8
7.4.5	C Language Options (<code>-c</code> and <code>-cr</code> Options)	7-9
7.4.6	Define an Entry Point (<code>-e global_symbol</code> Option)	7-9
7.4.7	Set Default Fill Value (<code>-f fill_value</code> Option)	7-10
7.4.8	Make a Symbol Global (<code>-g symbol</code> Option)	7-10
7.4.9	Make All Global Symbols Static (<code>-h</code> Option)	7-10
7.4.10	Define Heap Size (<code>-heap size</code> Option)	7-11
7.4.11	Alter the Library Search Algorithm (<code>-l</code> Option, <code>-L</code> Option, and <code>C_DIR</code> Environment Variable)	7-11
7.4.12	Disable Conditional Linking (<code>-j</code> Option)	7-13
7.4.13	Create a Map File (<code>-m filename</code> Option)	7-13
7.4.14	Name an Output Module (<code>-o filename</code> Option)	7-15
7.4.15	Strip Symbolic Information (<code>-s</code> Option)	7-15
7.4.16	Define Stack Size (<code>-stack size</code> Option)	7-16
7.4.17	Generate Far Call Trampolines (<code>--trampolines</code> Option)	7-16
7.4.18	Introduce an Unresolved Symbol (<code>-u symbol</code> Option)	7-18
7.4.19	Display a Message When an Undefined Output Section Is Created (<code>-w</code> Option)	7-19
7.4.20	Exhaustively Read and Search Libraries (<code>-x</code> and <code>-priority</code> Options)	7-19
7.4.21	Generate XML Link Information File (<code>--xml_link_info</code> Option)	7-20
7.5	Linker Command Files	7-21
7.5.1	Reserved Names in Linker Command Files	7-23
7.5.2	Constants in Linker Command Files	7-23
7.6	Object Libraries	7-24
7.7	The MEMORY Directive	7-26
7.7.1	Default Memory Model	7-26
7.7.2	MEMORY Directive Syntax	7-26

7.8	The SECTIONS Directive	7-30
7.8.1	SECTIONS Directive Syntax	7-30
7.8.2	Allocation	7-33
7.8.3	Specifying Input Sections	7-38
7.8.4	Allocation Using Multiple Memory Ranges	7-40
7.8.5	Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges	7-41
7.8.6	Allocating an Archive Member to an Output Section	7-43
7.9	Specifying a Section's Run-Time Address	7-44
7.9.1	Specifying Load and Run Addresses	7-44
7.9.2	Uninitialized Sections	7-45
7.9.3	Referring to the Load Address by Using the .label Directive	7-45
7.10	Using UNION and GROUP Statements	7-48
7.10.1	Overlaying Sections With the UNION Statement	7-48
7.10.2	Grouping Output Sections Together	7-50
7.11	Special Section Types (DSECT, COPY, and NOLOAD)	7-51
7.12	Default Allocation Algorithm	7-52
7.12.1	How the Allocation Algorithm Creates Output Sections	7-52
7.12.2	Reducing Memory Fragmentation	7-53
7.13	Assigning Symbols at Link Time	7-54
7.13.1	Syntax of Assignment Statements	7-54
7.13.2	Assigning the SPC to a Symbol	7-55
7.13.3	Assignment Expressions	7-55
7.13.4	Symbols Defined by the Linker	7-57
7.13.5	Assigning Exact Start, End, and Size Values of a Section to a Symbol	7-58
7.13.6	Why the Dot Operator Does Not Always Work	7-58
7.13.7	Address and Dimension Operators	7-59
7.14	Creating and Filling Holes	7-62
7.14.1	Initialized and Uninitialized Sections	7-62
7.14.2	Creating Holes	7-62
7.14.3	Filling Holes	7-64
7.14.4	Explicit Initialization of Uninitialized Sections	7-65
7.15	Linker-Generated Copy Tables	7-66
7.15.1	A Current Boot-Loaded Application Development Process	7-66
7.15.2	An Alternative Approach	7-67
7.15.3	Overlay Management Example	7-68
7.15.4	Generating Copy Tables Automatically with the Linker	7-69
7.15.5	The table() Operator	7-70
7.15.6	Boot-Time Copy Tables	7-70
7.15.7	Using the table() Operator to Manage Object Components	7-71
7.15.8	Copy Table Contents	7-72
7.15.9	General Purpose Copy Routine	7-73
7.15.10	Linker Generated Copy Table Sections and Symbols	7-77
7.15.11	Splitting Object Components and Overlay Management	7-78

7.16	Partial (Incremental) Linking	7-81
7.17	Linking C/C++ Code	7-83
7.17.1	Run-Time Initialization	7-83
7.17.2	Object Libraries and Run-Time Support	7-83
7.17.3	Setting the Size of the .stack and .system Sections	7-84
7.17.4	Autoinitialization of Variables at Run Time	7-84
7.17.5	Initialization of Variables at Load Time	7-85
7.17.6	The -c and -cr Linker Options	7-86
7.18	Linker Example	7-87
8	Absolute Lister Description	8-1
	<i>Explains how to invoke the absolute lister to obtain a listing of the absolute addresses of an object file.</i>	
8.1	Producing an Absolute Listing	8-2
8.2	Invoking the Absolute Lister	8-3
8.3	Absolute Lister Example	8-5
9	Cross-Reference Lister	9-1
	<i>Explains how to invoke the cross-reference lister to obtain a listing of symbols, their definitions, and their references in the linked source files.</i>	
9.1	Producing a Cross-Reference Listing	9-2
9.2	Invoking the Cross-Reference Lister	9-3
9.3	Cross-Reference Listing Example	9-4
10	Object File Utilities Descriptions	10-1
	<i>Explains how to invoke the object file display utility , the name utility, and the strip utility.</i>	
10.1	Invoking the Object File Display Utility	10-2
10.2	XML Tag Index	10-3
10.3	Example XML Consumer	10-9
10.3.1	The Main Application	10-9
10.3.2	xml.h Declaration of the XMLEntity Object	10-12
10.3.3	xml.cpp Definition of the XMLEntity Object	10-13
10.4	Invoking the Name Utility	10-16
10.5	Invoking the Strip Utility	10-17

11 Hex Conversion Utility Description 11-1

Explains how to invoke the hex utility to convert a COFF object file into one of several standard hexadecimal formats suitable for loading into an EPROM programmer.

11.1	The Hex Conversion Utility's Role in the Software Development Flow	11-2
11.2	Invoking the Hex Conversion Utility	11-3
11.2.1	Invoking the Hex Conversion Utility From the Command Line	11-3
11.2.2	Invoking the Hex Conversion Utility With a Command File	11-6
11.3	Understanding Memory Widths	11-8
11.3.1	Target Width	11-9
11.3.2	Specifying the Memory Width	11-9
11.3.3	Partitioning Data Into Output Files	11-10
11.4	The ROMS Directive	11-13
11.4.1	When to Use the ROMS Directive	11-15
11.4.2	An Example of the ROMS Directive	11-15
11.5	The SECTIONS Directive	11-19
11.6	Assigning Output Filenames	11-20
11.7	Image Mode and the -fill Option	11-22
11.7.1	Generating a Memory Image	11-22
11.7.2	Specifying a Fill Value	11-23
11.7.3	Steps to Follow in Using Image Mode	11-23
11.8	Controlling the ROM Device Address	11-24
11.9	Description of the Object Formats	11-25
11.9.1	ASCII-Hex Object Format (-a Option)	11-26
11.9.2	Intel MCS-86 Object Format (-i Option)	11-27
11.9.3	Motorola Exorciser Object Format (-m Option)	11-28
11.9.4	Texas Instruments SDSMAC Object Format (-t Option)	11-29
11.9.5	Extended Tektronix Object Format (-x Option)	11-30

A Common Object File Format A-1

Contains supplemental technical data about the internal format and structure of COFF object files.

A.1	COFF File Structure	A-2
A.2	File Header Structure	A-4
A.3	Optional File Header Format	A-6
A.4	Section Header Structure	A-7
A.5	Structuring Relocation Information	A-10
A.6	Symbol Table Structure and Content	A-12
A.6.1	Special Symbols	A-13
A.6.2	Symbol Name Format	A-14
A.6.3	String Table Structure	A-14
A.6.4	Storage Classes	A-15
A.6.5	Symbol Values	A-15
A.6.6	Section Number	A-16
A.6.7	Auxiliary Entries	A-16

B	Symbolic Debugging Directives	B-1
	<i>Discusses symbolic debugging directives that the TMS470R1x C/C++ compiler uses.</i>	
B.1	DWARF Debugging Format	B-2
B.2	COFF Debugging Format	B-3
B.3	Debug Directive Syntax	B-4
C	XML Link Information File Description	C-1
	<i>Discusses the xml_link_info file contents including file element types and document elements.</i>	
C.1	XML Information File Element Types	C-2
C.2	Document Elements	C-3
C.2.1	Header Elements	C-3
C.2.2	Input File List	C-4
C.2.3	Object Component List	C-5
C.2.4	Logical Group List	C-6
C.2.5	Placement Map	C-9
C.2.6	Far Call Trampoline List	C-11
C.2.7	Symbol Table	C-12
D	Hex Conversion Utility Examples	D-1
	<i>Demonstrates command-file development for a variety of memory systems and situations.</i>	
D.1	Scenario 1: Building a Hex Conversion Command File for a Single 8-Bit EPROM	D-2
D.2	Scenario 2: Building a Hex Conversion Command File for 16-BIS Code	D-8
D.3	Scenario 3: Building a Hex Conversion Command File for Two 8-Bit EPROMs	D-12
E	Glossary	E-1
	<i>Defines terms and acronyms used in this book.</i>	

Figures

1-1	TMS470R1x Software Development Flow	1-2
2-1	Partitioning Memory Into Logical Blocks	2-3
2-2	Object Code Generated by the File in Example 2-1	2-10
2-3	Combining Input Sections to Form an Executable Object Module	2-12
3-1	The Assembler in the TMS470R1x Software Development Flow	3-3
4-1	The .space and .bes Directives	4-11
4-2	The .field Directive	4-12
4-3	Initialization Directives	4-14
4-4	The .align Directive	4-16
4-5	Double-Precision Floating-Point Format	4-36
4-6	The .field Directive	4-43
4-7	Single-Precision Floating-Point Format	4-44
4-8	The .usect Directive	4-78
6-1	The Archiver in the TMS470R1x Software Development Flow	6-3
7-1	The Linker in the TMS470R1x Software Development Flow	7-3
7-2	Section Allocation Defined by Example 7-4	7-33
7-3	Run-Time Execution of Example 7-6	7-47
7-4	Memory Allocation Shown in Example 7-7 and Example 7-8	7-49
7-5	Autoinitialization at Run Time	7-84
7-6	Autoinitialization at Load Time	7-85
8-1	The Absolute Lister in the TMS470R1x Software Development Flow	8-2
9-1	The Cross-Reference Lister in the TMS470R1x Software Development Flow	9-2
11-1	The Hex Conversion Utility in the TMS470R1x Software Development Flow	11-2
11-2	Hex Conversion Utility Process Flow	11-8
11-3	COFF Data and Memory Widths	11-10
11-4	Data, Memory, and ROM Widths	11-12
11-5	The infile.out File Partitioned Into Four Output Files	11-16
11-6	ASCII-Hex Object Format	11-26
11-7	Intel Hexadecimal Object Format	11-27
11-8	Motorola-S Object Format	11-28
11-9	TI-Tagged Object Format	11-29
11-10	Extended Tektronix Object Format	11-30
A-1	COFF File Structure	A-2
A-2	Sample COFF Object File	A-3
A-3	Section Header Pointers for the .text Section	A-9
A-4	Symbol Table Contents	A-12
A-5	String Table Entries for Sample Symbol Names	A-14
D-1	EPROM Memory System for Scenario 1	D-2
D-2	EPROM Memory System for Scenario 2	D-8
D-3	EPROM Memory System for Scenario 3	D-12

Tables

3-1	Operators Used in Expressions (Precedence)	3-27
3-2	Expressions With Absolute and Relocatable Symbols	3-29
3-3	Symbol Attributes	3-38
4-1	Assembler Directives Summary	4-2
5-1	Substitution Symbol Functions and Return Values	5-8
5-2	Creating Macros	5-23
5-3	Manipulating Substitution Symbols	5-23
5-4	Conditional Assembly	5-23
5-5	Producing Assembly-Time Messages	5-24
5-6	Formatting the Listing	5-24
7-1	Linker Options Summary	7-5
7-2	Groups of Operators Used in Expressions (Precedence)	7-56
9-1	Symbol Attributes	9-5
10-1	XML Tag Index	10-3
11-1	Hex Conversion Utility Options	11-4
11-2	Options for Specifying Hex Conversion Formats	11-25
A-1	File Header Contents	A-4
A-2	File Header Flags (Bytes 18 and 19)	A-5
A-3	Optional File Header Contents	A-6
A-4	Section Header Contents	A-7
A-5	Section Header Flags (Bytes 36 Through 39)	A-8
A-6	Relocation Entry Contents	A-10
A-7	Relocation Types (Bytes 8 and 9)	A-11
A-8	Symbol Table Entry Contents	A-13
A-9	Special Symbols in the Symbol Table	A-13
A-10	Symbol Storage Classes	A-15
A-11	Section Numbers	A-16
A-12	Section Format for Auxiliary Table Entries	A-16
B-1	Symbolic Debugging Directives	B-4

Examples

2-1	Using Sections Directives	2-9
2-2	Code That Generates Relocation Entries	2-14
3-1	Local Labels of the Form \$n	3-20
3-2	Using Symbolic Constants Defined on the Command-Line	3-22
3-3	Assembler Listing	3-33
3-4	Viewing Assembly Variables as C Types	3-35
3-5	Assembler Cross-Reference Listing	3-37
4-1	Sections Directives	4-9
5-1	Macro Definition, Call, and Expansion	5-4
5-2	Calling a Macro With Varying Numbers of Arguments	5-6
5-3	The .asg Directive	5-6
5-4	The .eval Directive	5-7
5-5	Using Built-In Substitution Symbol Functions	5-8
5-6	Recursive Substitution	5-9
5-7	Using the Forced Substitution Operator	5-10
5-8	Using Subscripted Substitution Symbols to Redefine an Instruction	5-11
5-9	Using Subscripted Substitution Symbols to Find Substrings	5-12
5-10	The .loop/.break/.endloop Directives	5-15
5-11	Nested Conditional Assembly Directives	5-15
5-12	Built-In Substitution Symbol Functions Used in a Conditional Assembly Code Block	5-15
5-13	Unique Labels in a Macro	5-16
5-14	Producing Messages in a Macro	5-18
5-15	Using Nested Macros	5-21
5-16	Using Recursive Macros	5-22
7-1	Linker Command File	7-21
7-2	Command File With Linker Directives	7-22
7-3	The MEMORY Directive	7-27
7-4	The SECTIONS Directive	7-32
7-5	The Most Common Method of Specifying Section Contents	7-38
7-6	Copying a Section From EXT_MEM to P_MEM	7-46
7-7	The UNION Statement	7-48
7-8	Separate Load Addresses for UNION Sections	7-48
7-9	Allocate Sections Together	7-50
7-10	Default Allocation for TMS470R1x Devices	7-52
7-11	Using a UNION for Memory Overlay	7-68

7-12	Produce Address for Linker Generated Copy Table	7-69
7-13	Linker Command File to Manage Object Components	7-71
7-14	TMS470 cpy_tbl.h File	7-72
7-15	Run-Time-Support cpy_tbl.c File	7-74
7-16	The cpy_utils.asm File	7-75
7-17	Controlling the Placement of the Linker-Generated Copy Table Sections	7-77
7-18	Creating a Copy Table to Access a Split Object Component	7-79
7-19	Split Object Component Driver	7-79
7-20	Linker Command File, demo.cmd	7-88
7-21	Output Map File, demo.map	7-89
11-1	A ROMS Directive Example	11-16
11-2	Map File Output From Example 11-1 That Shows Memory Ranges	11-18
C-1	Header Element for the hi.out Output File	C-3
C-2	Input File List for the hi.out Output File	C-4
C-3	Object Component List for the fl-4 Input File	C-5
C-4	Logical Group List for the fl-4 Input File	C-8
C-5	Placement Map for the fl-4 Input File	C-10
C-6	Fall Call Trampoline List for the fl-4 Input File	C-12
C-7	Symbol Table for the fl-4 Input File	C-13
D-1	Assembly Code for Hex Conversion Utility Scenarios	D-1
D-2	Linker Command File and Link Map for Scenario 1	D-3
D-3	Hex Conversion Command File for Scenario 1	D-5
D-4	Contents of Hex Map File example1.mxp	D-6
D-5	Contents of Hex Output File example1.hex	D-7
D-6	Linker Command File for Scenario 2	D-9
D-7	Hex Conversion Command File for Scenario 2	D-10
D-8	Contents of Hex Map File example2.mxp	D-11
D-9	Contents of Hex Output File, example2.hex	D-11
D-10	Linker Command File for Scenario 3	D-13
D-11	Hex Conversion Command File for Scenario 3	D-15
D-12	Contents of Hex Map File example3.mxp	D-15
D-13	Contents of Hex Output File lower16.bit	D-16
D-14	Contents of Hex Output File upper16.bit	D-16

Notes

Default Placement by the Assembler	2-4
Labels and Comments Not Shown in Syntaxes	4-2
The .byte, .char, .word, .int, .long, .string, .double, .float, .half, .short, and .field Directives in a .struct/.endstruct Sequence	4-13
Ending a Macro	4-39
Directives That Can Appear in a .struct/.endstruct Sequence	4-71
Naming Library Members	6-5
–a and –r Options	7-6
Filling Memory Ranges	7-28
Binding is Incompatible with Alignment and Named Memory	7-35
You Cannot Specify Addresses for Sections Within a GROUP	7-50
Linker Command File Operator Equivalencies	7-59
Filling Sections	7-65
The TI-Tagged Format Is 16 Bits Wide	11-11
Sections Generated by the C Compiler	11-19
Defining the Ranges of Target Memory	11-22

Introduction to the Software Development Tools

The TMS470R1x™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities. This chapter provides an overview of these tools.

The TMS470R1x is supported by the following assembly language development tools:

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

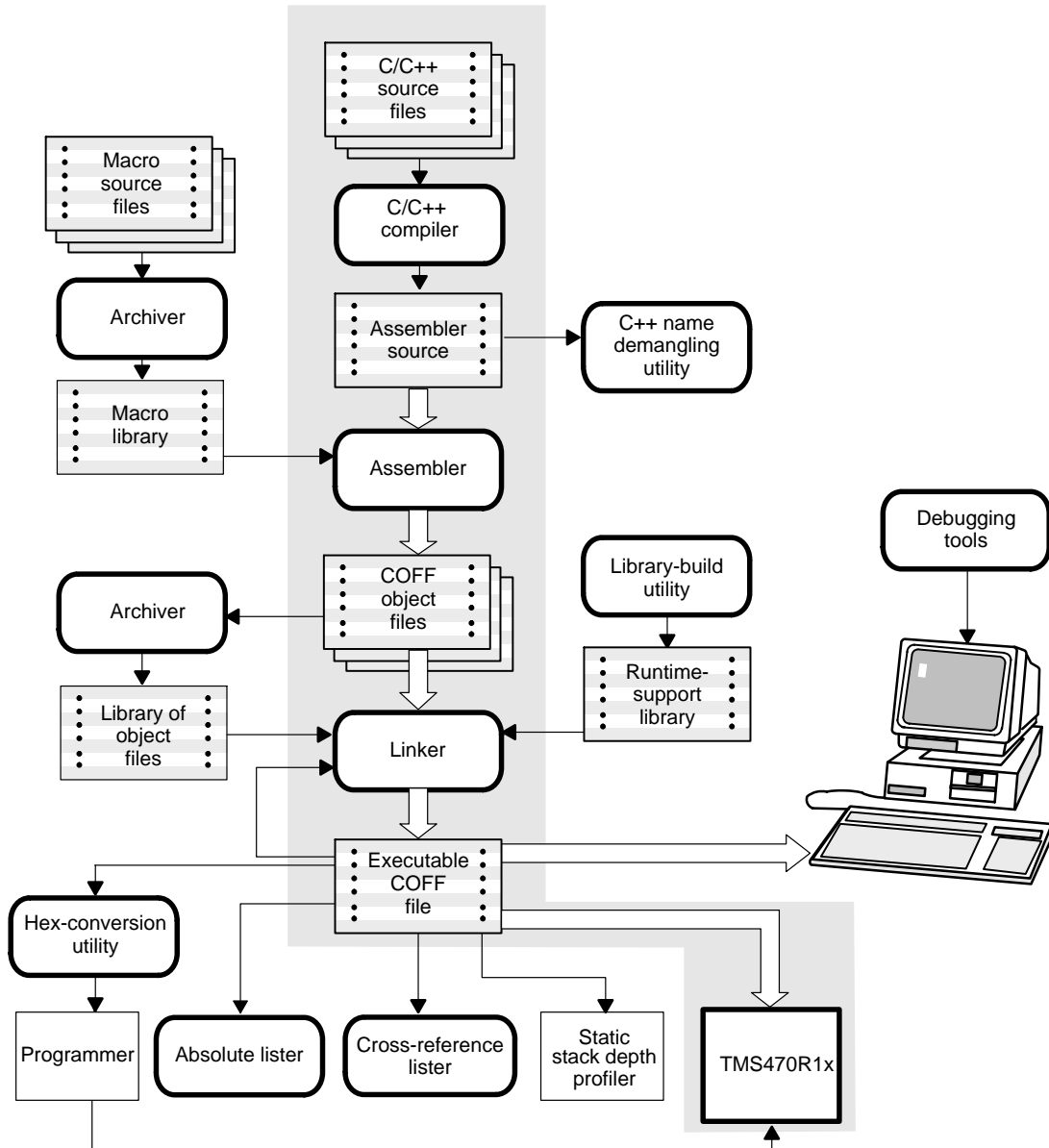
This chapter shows how these tools fit into the general software tools development flow and gives a brief description of each tool. For convenience, it also summarizes the C compiler and debugging tools. For detailed information on the compiler and the debugger, and for complete descriptions of the TMS470R1x, see the books listed in *Related Documentation From Texas Instruments* on page vi.

Topic	Page
1.1 Software Development Tools Overview	1-2
1.2 Tools Descriptions	1-3

1.1 Software Development Tools Overview

Figure 1–1 shows the TMS470R1x 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. TMS470R1x Software Development Flow



1.2 Tools Descriptions

The following list describes the tools that are shown in Figure 1–1:

- ❑ The **C compiler** accepts C source code and produces TMS470R1x 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 source statements with assembly language output to correlate code produced by the compiler with your source code.

For more information, see the *TMS470R1x Optimizing C Compiler User's Guide*.

- ❑ 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. For more information, see Chapter 3, *Assembler Description*, through Chapter 5, *Macro Language*. See the *TMS470R1x User's Guide* for detailed information on the 16-bit and 32-bit assembly language instruction sets.
- ❑ 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 runtime-support library. See the *TMS470R1x Optimizing C 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** is a debugging tool. It accepts linked object files as input and creates *.abs* files as output. These files can be assembled to produce a listing that contains absolute rather than relative addresses. Without the absolute lister, producing such a listing would be tedious and require many manual operations. 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 **TMS470R1x** 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 extended development system (XDS510) emulator

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

Introduction to Common Object File Format

The assembler and linker create object files that can be executed by a TMS470R1x 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 on COFF object file structure.

Topic	Page
2.1 Sections	2-2
2.2 How the Assembler Handles Sections	2-4
2.3 How the Linker Handles Sections	2-11
2.4 Relocation	2-13
2.5 Run-Time Relocation	2-15
2.6 Loading a Program	2-16
2.7 Symbols in a COFF File	2-17

2.1 Sections

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

.text section	usually contains executable code
.data section	usually contains initialized data
.bss section	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:

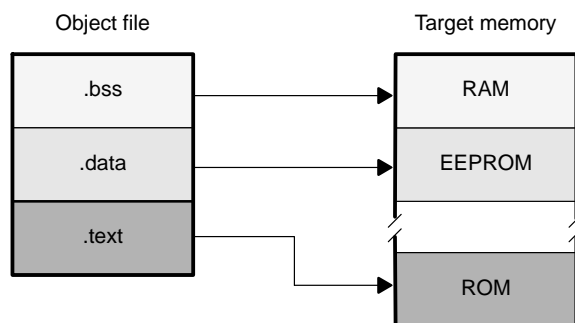
Initialized sections	contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.
Uninitialized sections	reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created 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 place 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*, on page 2-7, for more information.

Note: Default Placement by the Assembler

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 TMS470R1x 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 syntax for these directives is:

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

<i>symbol</i>	points to the first word reserved by this invocation of the directive. The <i>symbol</i> corresponds to the name of the variable for which you are reserving space. It can be referenced by any other section and can also be declared as a global symbol (with the <code>.global</code> assembler directive).
<i>size in bytes</i>	is an absolute expression. <ul style="list-style-type: none"> <input type="checkbox"/> The <code>.bss</code> directive reserves the specified number of bytes in the <code>.bss</code> section. The default <i>size in bytes</i> is 1 byte. <input type="checkbox"/> The <code>.usect</code> directive reserves specified number of bytes in <i>section name</i>. You must specify a size; there is no default size.
<i>alignment</i>	is an optional parameter. It specifies the minimum alignment in bytes required by the space allocated.
<i>section name</i>	tells the assembler which named section to reserve space in. For more information, see subsection 2.2.3, <i>Named Sections</i> , on page 2-6.

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, *An Example That Uses Sections Directives*, on page 2-8.

The assembler treats uninitialized subsections (created with the `.usect` directive) in the same manner as uninitialized sections. See section 2.2.4, *Subsections*, on page 2-7 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 TMS470R1x memory when the program is loaded. Each initialized section is independently relocatable and can 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 using 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 contiguously 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, *Subsections*, on page 2-7 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]
.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, *Subsections*, 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 syntax for creating a subsection is:

```
symbol .usect "section name:subsection name", size in bytes [, alignment]
.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*, on page 7-30, for an example using subsections.

You can create two types of subsections:

- ☐ Initialized subsections are created using the `.sect` directive. See subsection 2.2.2, *Initialized Sections*, on page 2-6.
- ☐ Uninitialized subsections are created using the `.usect` directive. See subsection 2.2.1, *Uninitialized Sections*, on page 2-4.

Subsections are allocated in the same manner as sections. See section 7.8, *The SECTIONS Directive*, on page 7-30, for more information.

2.2.5 Section Program Counters (SPCs)

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*, on page 2-13.

2.2.6 An Example That Uses Sections Directives

Example 2-1 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 Example 2-1 is a listing file. Example 2-1 shows how the SPCs are modified during assembly. A line in a listing file has four fields:

- Field 1** contains the source code line counter.
- Field 2** contains the section program counter.
- Field 3** contains the object code.
- Field 4** contains the original source statement.

Example 2-1. Using Sections Directives

```

1          ****
2          ** Assemble an initialized table into .data. **
3          ****
4 00000000          .data
5 00000000 00000011 coeff  .word          011h, 022h, 033h
   00000004 00000022
   00000008 00000033
6
7          ****
8          ** Reserve space in .bss for a variable. **
9          ****
10 00000000          .bss          buffer,10
11          ****
12          ** Still in .data. **
13          ****
14 0000000c 00000123 ptr  .word          0123h
15          ****
16          ** Assemble code into the .text section. **
17          ****
18 00000000          .text
19 00000000 E59F14D2 add:   LDR          R1, #1234
20 00000004 E2511001 aloop: SUBS          R1, R1, #1
21 00000008 1AFFFFFD          BNE          aloop
22          ****
23          ** Another initialized table into .data. **
24          ****
25 00000010          .data
26 00000010 000000AA ivals  .word          0AAh, 0BBh, 0CC
   00000014 000000BB
   00000018 000000CC
27          ****
28          ** Define another section for more variables.**
29          ****
30 00000000          var2  .usect          "newvars", 1
31 00000001          inbuf .usect          "newvars", 7
32          ****
33          ** Assemble more code into .text. **
34          ****
35 0000000c          .text
36 0000000c E59F3D80 mpy:   LDR          R3, #3456
37 00000010 E0120293 mloop: MULS          R2, R3, R2
38 00000014 1AFFFFFD          BNE          mloop
39          ****
40          ** Define a named section for int. vectors. **
41          ****
42 00000000          .sect          "vectors"
43 00000000 00000011          .word          011h,033h
   00000004 00000033

```

Field 1 Field 2 Field 3 Field 4

As Figure 2–2 shows, the file in Example 2–1 creates five sections:

.text	contains six 32-bit words of object code.
.data	contains seven words of object code.
vectors	is a named section created with the <code>.sect</code> directive; it contains two words of initialized data.
.bss	reserves ten bytes in memory.
newvars	is a named section created with the <code>.usect</code> directive; it reserves 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 2–2. Object Code Generated by the File in Example 2–1

Line numbers	Object code	Section
19 20 21 36 37 38	E59F14D2 E2511001 1AFFFFFFD E59F3D80 E0120293 1AFFFFFFD	.text
5 5 5 14 26 26 26	00000011 00000022 00000033 00000123 000000AA 000000BB 000000CC	.data
43 43	00000011 00000033	vectors
10	No data — ten bytes reserved	.bss
30 31	No data — eight 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. When you *do* use linker directives, you must specify them in a linker command file.

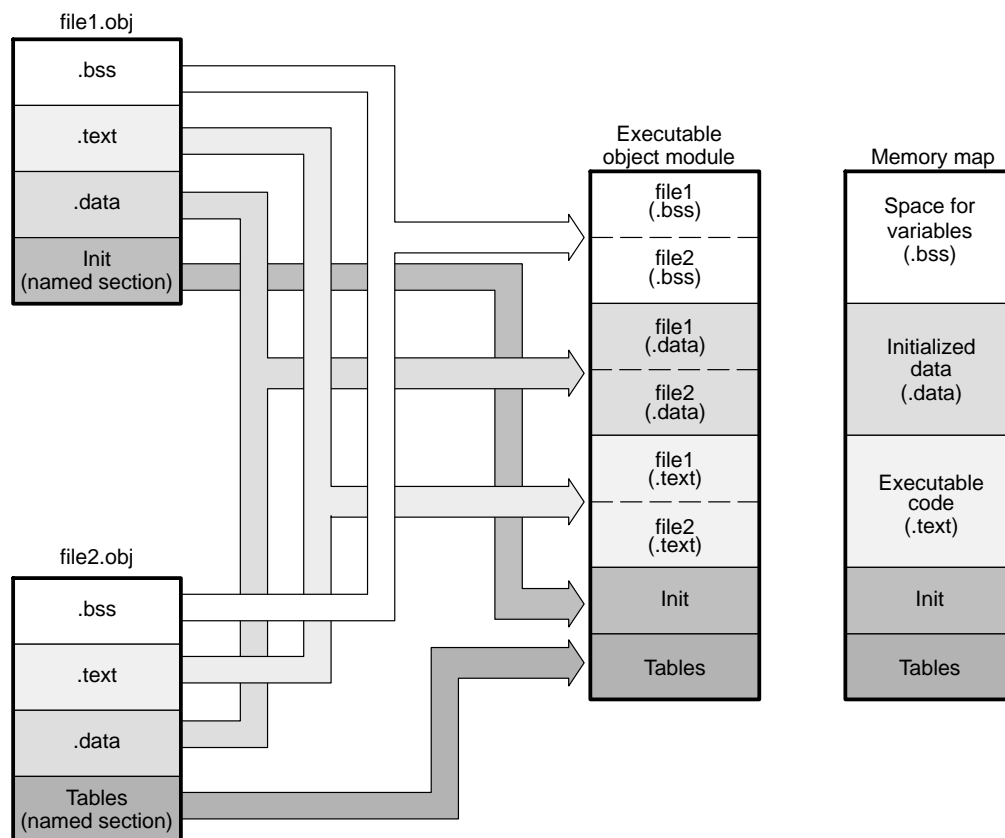
See the following sections for more information about linker command files and linker directives:

Section	Page
7.5 Linker Command Files	7-21
7.7 The MEMORY Directive	7-26
7.8 The SECTIONS Directive	7-30
7.12 Default Allocation Algorithm	7-52

2.3.1 Default Memory Allocation

Figure 2–3 illustrates the process of linking two files together.

Figure 2–3. Combining Input Sections to Form an Executable Object Module



In Figure 2–3, 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 with 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 address 0h and places the sections one after the other as shown.

2.3.2 Placing Sections in the Memory Map

Figure 2–3 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*, on page 7-26, and section 7.8, *The SECTIONS Directive*, on page 7-30.

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 2–2 contains a code segment for the TMS470R1x that generates relocation entries.

Example 2–2. Code That Generates Relocation Entries

```

1          *****
2          **      Generating Relocation Entries      **
3          *****
4          .ref X
5          .def Y
6 00000000 .text
7 00000000 E0921003     ADDS    R1, R2, R3
8 00000004 0A000001     BEQ     Y
9 00000008 E1C410BE     STRH    R1, [R4, #14]
10 0000000c EAffFFFFB!   B      X      ; generates a relocation entry
11 00000010 E0821003 Y:   ADD     R1, R2, R3

```

In Example 2–2, 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 a relocation entries for X. The reference to X is an external reference (indicated by the ! character in the listing).

After the code is linked, suppose that X is relocated to address 10014h. Suppose also that the .text section is relocated to begin at address 10000h; Y now has a relocated value of 10010h. The linker uses the relocation entry for the reference to X to patch the branch instruction in the object code:

EAffFFFFB! B X becomes EA000000

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 page 7-6).

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 a ROM-based system. The code must be loaded into RAM, but it would run faster in ROM.

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 7-6, on page 7-46.

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 different 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*, on page 7-44.

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. Two common situations are described here.

- ☐ The TMS470R1x debugging tools, including the software simulator and the XDS™ emulator have built-in loaders. Each of these tools contains a LOAD command that invokes a loader; the loader reads the executable file and copies the program into target memory.
- ☐ You can use the hex conversion utility (hex470, 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.

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:

- `.def`** The symbol is defined in the current module and used in another module.
- `.ref`** The symbol is referenced in the current module, but defined in another module.
- `.global`** The symbol can be either of the above.

The following code segment illustrates these definitions.

```
x:  ADD      R0, #56h      ; Define x
    B        y            ; Reference y
    .global  x            ; .def of x
    .global  y            ; .ref of y
```

The `.global` directive for `x` declares that it is an external symbol defined in this module and that other modules can reference `x`. The `.global` directive for `y` declares that it is an undefined symbol that is defined in another module. The assembler determines that `y` is defined in another module because it is not defined in the current module.

The assembler places both `x` and `y` in the object file's symbol table. When the file is linked with other object files, the entry for `x` resolves references to `x` in other files. The entry for `y` causes the linker to look through the symbol tables of other files for `y`'s definition.

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 on page 2-17). 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 in a section.

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 symbolic debugging 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 page 3-5).

Assembler Description

The 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 Chapter 4
Macro directives	described in Chapter 5
Assembly language instructions	described in the <i>TMS470R1x User's Guide</i>

This chapter explains how to invoke the assembler and discusses source statement format, valid constants and expressions, and assembler output.

Topic	Page
3.1 Assembler Overview	3-2
3.2 The Assembler's Role in the Software Development Flow	3-3
3.3 Invoking the Assembler	3-4
3.4 Naming Alternate Directories for Assembler Input	3-7
3.5 Source Statement Format	3-10
3.6 Constants	3-16
3.7 Character Strings	3-18
3.8 Symbols	3-19
3.9 Expressions	3-26
3.10 Source Listings	3-31
3.12 Cross-Reference Listings	3-37

3.1 Assembler Overview

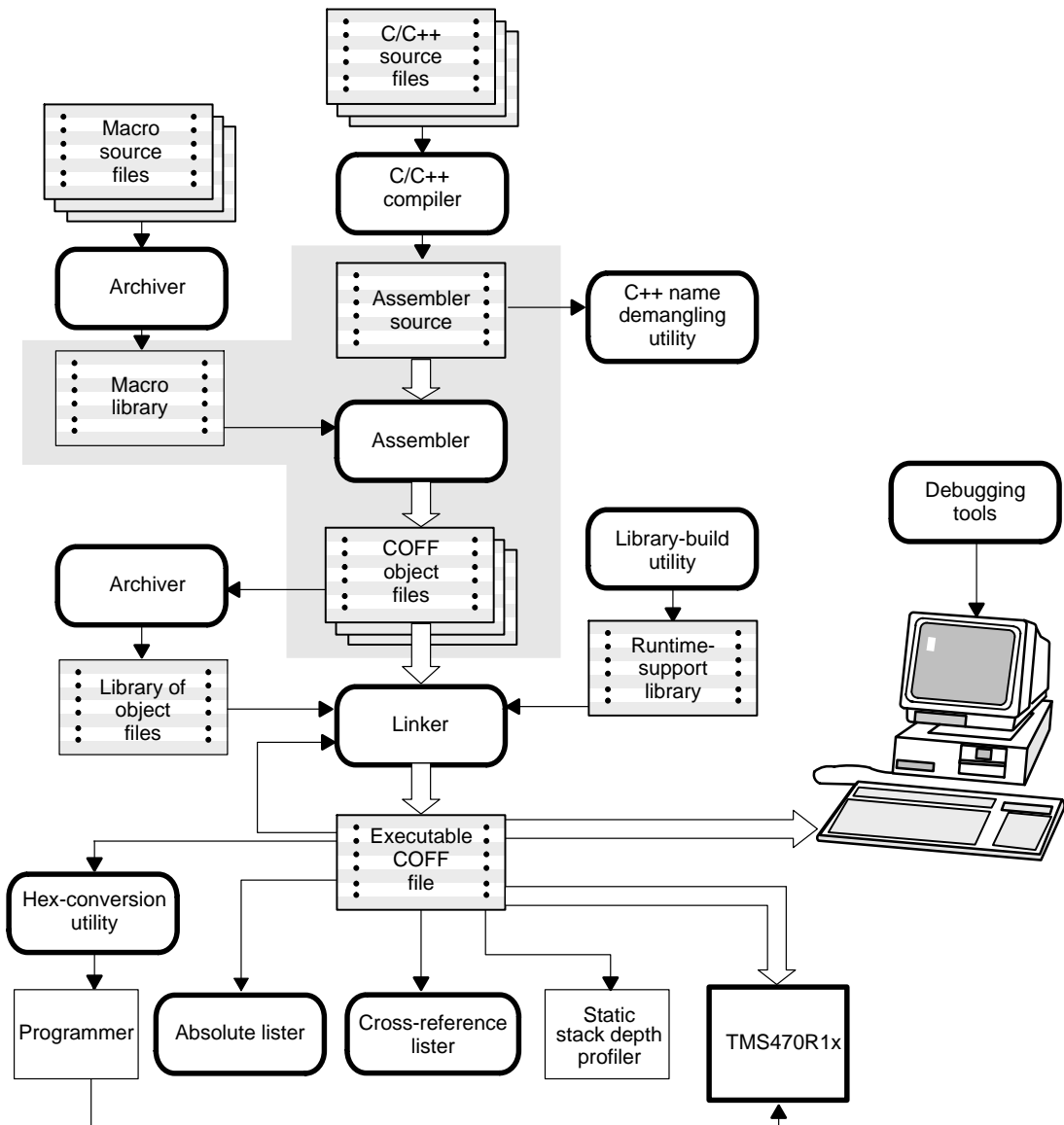
The two-pass assembler does the following:

- ☐ 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 an SPC (section program counter) 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 TMS470R1x C compiler.

Figure 3–1. The Assembler in the TMS470R1x Software Development Flow



3.3 Invoking the Assembler

To invoke the assembler, enter the following:

cl470 *input file* [*options*]

- cl470** 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* identifies 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. Options must be specified separately.

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.
- ahc** **-ahcfilename** 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** **-ahifilename** 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.
- as** 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.
- ax** 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.
- d** *-dname [=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 (-d Option)*, on page 3-21.
- f** prevents the automatic appending of *.asm* as the extension for assembly files.
- g** 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 section 3.11, *Debugging Assembly Source*, on page 3-35 for more information.
- I** 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 section 3.4.1, *Using the -I Assembler Option*, on page 3-7.
- me** produces object code in little-endian format.

- mt** instructs the assembler to begin assembling instructions as 16-bit instructions. By default, the assembler begins assembling 32-bit instructions.
- q** suppresses the banner and all progress information (assembler runs in quiet mode).
- u** **-u***name* undefines the predefined constant *name*, which overrides any **-d** options for the specified constant.

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 set with the `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 on page 3-7) or the `A_DIR` environment variable (described in section 3.4.2 on page 3-8).

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:

```
cl470 -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	<code>cl470 -I/470tools/files source.asm</code>
Windows	<code>cl470 -Ic:\470tools\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 A_DIR Environment Variable

An environment variable is a system symbol that you define and assign a string to. The assembler uses the `A_DIR` environment variable to name alternate directories that contain copy/include files or macro libraries.

If the assembler does not find `A_DIR`, it then searches for `C_DIR`. See the *TMS470R1x Optimizing Compiler User's Guide* for details on `C_DIR`.

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

Operating System	Enter
Windows	<code>set A_DIR= pathname₁;pathname₂; . . .</code>
UNIX (Bourne shell)	<code>A_DIR="pathname₁;pathname₂; . . ."; export A_DIR</code>

The *pathnames* are directories that contain copy/include files or macro libraries. You can separate the pathnames with a semicolon or with blanks. 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: /470tools/files/copy1.asm
 /dsys/copy2.asm

Windows: c:\470tools\files\copy1.asm
 c:\dsys\copy2.asm

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

Operating System	Enter
UNIX (Bourne Shell)	<code>A_DIR="/dsys"; export A_DIR</code> <code>cl470 -I=/470tools/files source.asm</code>
Windows	<code>set A_DIR=c:\dsys</code> <code>cl470 -Ic:\470tools\files source.asm</code>

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.

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

Operating System	Enter
Windows	<code>set A_DIR=</code>
UNIX	<code>unset A_DIR</code>

3.5 Source Statement Format

TMS470R1x assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. Each source statement can contain four ordered fields (label, mnemonic, operand list, and comment). The general syntax for source statements is as follows:

[label] [:] mnemonic [operand list] [:comment]

The following are examples of source statements:

```
SYM1      .set      2           ; Symbol SYM1 = 2.
Begin:    MOV       R0, #SYM1   ; Load R0 with 2.
          .word     016h        ; Initialize word (016h).
```

The TMS470 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.
- ☐ 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.

When you use a label, its value is the current value of the section program counter (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 would point to the first word. In the following example, the label `Start` has the value `40h`.

```

.      .      .      .
.      .      .      .
.      .      .      .
9 00000000      ; Assume some other 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 .set $ ; $ 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):

```

3 00000050      Here:
4 00000050 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 in the source statement. The mnemonic field cannot start in column 1; if it does, it is interpreted as a label. The mnemonic field can contain one of the following items:

- ☐ Machine-instruction mnemonic (such as `ADD`, `MUL`, `STR`)
- ☐ Assembler directive (such as `.data`, `.list`, `.set`)
- ☐ Macro directive (such as `.macro`, `.var`, `.mexit`)
- ☐ Macro call

3.5.3 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:

- ☐ Constants (see section 3.6 on page 3-16)
- ☐ Symbols (see section 3.8 on page 3-19)
- ☐ Expressions, a combination of constants and symbols (see section 3.9 on page 3-26)

You must separate operands with commas. The following sections describe the syntaxes for operands of instructions and the use of immediate values (constants) with directives.

3.5.3.1 Operand Syntaxes for Instructions

The assembler allows you to specify that an operand should be used as an address, an immediate value, an indirect address, a register, a shifted register, or a register list. The following rules apply to the operands of instructions.

- ☐ **# prefix — the operand is an immediate value.** If you use the # sign as a prefix, the assembler treats the operand as an immediate value. This is true even when the operand is a register; the assembler treats the register as a value instead of using the contents of the register. These are examples of an instruction that uses an operand with the # prefix:

```
Label:  ADD R1, R1, #123
        ; Add 123 (decimal) to the value of R1
        ; and place the result in R1.
```

- ☐ **Square brackets — the operand is an indirect address.** If the operand is enclosed in square brackets, the assembler treats the operand as an indirect address; that is, it uses the contents of the operand as an address.

Indirect addresses consist of a base and an offset. The base is specified by a register and is formed by taking the value in the register. The offset can be specified by a register, an immediate value, or a shifted register. Furthermore, the offset can be designated as one of the following:

- *Pre-index*, where the base and offset are combined to form the address. To designate a pre-index offset, include the offset within the enclosing right bracket.
- *Postindex*, where the address is formed from the base, and then the base and offset are combined. To designate a postindex offset, include the offset outside of the right bracket.

The offset can be added to or subtracted from the base.

The following are examples of instructions that use indirect addresses as operands:

```
A: LDR R1, [R1]
    ; Load from address in R1 into R1.
    LDR R7, [R1, #5]
    ; Form address by adding the value in R1 to 5. Load
    ; from address into R7.
    STR R3, [R1, -R2]
    ; Form address by subtracting the value in R2 from
    ; the value in R1. Store from R3 to memory at address.
    STR R14, [R1, +R3, LSL #2]
    ; Form address by adding the value in R3 shifted
    ; left by 2 to the value in R1. Store from R14
    ; to memory at address.
    LDR R1, [R1], #5
    ; Load from address in R1 into R1, then add 5 to the
    ; address.
    STR R2, [R1], R5
    ; Store value in R2 into the address in R1, then add
    ; the value in R5 to the address.
```

- ❑ **! suffix — writeback to register.** If you use the ! sign as a suffix, the assembler writes the computed address back to the base register. Writeback to register is used only with the indirect addressing mode syntax.

This is an example of an instruction using the writeback to register suffix:

```
LDR R1, [R4, #4]!
    ; Form address by adding the value in R4 to 4. Load
    ; from this address into R1, then replace the value
    ; in R4 with the address.
```

- ❑ **^ suffix — set S bit.** If you use the ^ sign as a suffix, the assembler sets the S bit. The resulting action depends on the type of instruction being executed and whether R15 is in the transfer list. For more information, see the LDM and STM instructions in the *TMS470R1x User's Guide*.

This is an example of an instruction using the set S bit suffix:

```
LDMIA SP, {R4-R11, R15}^
    ; Load registers R4 through R11 and R15 from memory
    ; at SP. Load CPSR with SPSR.
```

- ❑ **Shifted registers.** If a register symbol is followed by a shift type, the computed value is the value in the register shifted according to the type as defined below:

LSL – Logical shift left
LSR – Logical shift right
ASL – Arithmetic shift left
ASR – Arithmetic shift right
ROR – Rotate right
RRX – Rotate right extended

The shift type can be followed by a register or an immediate whose value defines the shift amount. The following are examples of instructions that use shifted registers as operands:

```
B: ADD R1, R4, R5, LSR R2
    ; Logical shift right the value in R5 by the value in
    ; R2. Add the value in R5 to R4. Place result in R1.
    LDR R1, [R5, R4, LSL #4]
    ; Form address by adding the value in R4 shifted left
    ; by 4 to the value in R5. Load from address into R1.
    CMP R3, R4, RRX
    ; Compare the value in R3 with the value in R4 rotate
    ; right extend.
```

- ❑ **Curly braces — the operand is a register list.** If you surround registers with curly braces, the assembler treats the operand as a list of registers. You can separate registers with commas or indicate a range of registers with a dash. The following are examples of instructions that use register lists:

```
LDMEA R2, {R1, R3, R6}
    ; Pre-decrement stack load. Load registers R1, R3
    ; and R6 from memory at the address in R2.
STMFd R12, {R1, R3-R5}
    ; Pre-increment stack store. Store from registers R1
    ; and R3 through R5 to memory at the address in R12.
```

3.5.3.2 Immediate Values as Operands for Directives

You use immediate values as operands primarily with instructions. In some cases, you can use immediate values with the operands of directives. For instance, you can use immediate values with the `.byte` directive to load values into the current section.

It is not usually necessary to use the `#` prefix for directives. Compare the following statements:

```
ADD R1, #10  
  
.byte 10
```

In the first statement, the `#` prefix is necessary to tell the assembler to add the value 10 to R1. In the second statement, however, the `#` prefix is not used; the assembler expects the operand to be a value and initializes a byte with the value 10.

See Chapter 4, *Assembly Directives*, for more information on the syntax and usage of directives.

3.5.4 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 0FFh is equal to 00FF (base 16) or 255 (base 10); it *does not* equal -1 . Note, however, that 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 16 binary digits (0s and 1s) followed by the suffix B (or b). If fewer than 16 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_{10} or 0_{16}
0100000b	Constant equal to 32_{10} or 20_{16}
01b	Constant equal to 1_{10} or 1_{16}
11111000B	Constant equal to 248_{10} or $0F8_{16}$

3.6.2 Octal Integers

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

10Q	Constant equal to 8_{10} or 8_{16}
100000Q	Constant equal to $32\,768_{10}$ or 8000_{16}
226q	Constant equal to 150_{10} or 96_{16}

3.6.3 Decimal Integers

A decimal integer constant is a string of decimal digits ranging from $-32\,768$ to $65\,535$. These are examples of valid decimal constants:

1000	Constant equal to 1000_{10} or $3E8_{16}$
-32768	Constant equal to $-32\,768_{10}$ or 8000_{16}
25	Constant equal to 25_{10} or 19_{16}

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 and a–f. A hexadecimal constant must begin with a decimal value (0–9). If fewer than four 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}
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 string of 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 (see page 4-63) to assign a value to a symbol, the symbol becomes a constant. To use this constant in expressions, the value that is assigned to it must be absolute. For example:

```
shift3      .set    3
            MOV     R0, #shift3
```

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:

```
AuxR1      .set    R1
            LDR     AuxR1, [SP]
```

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 page 3-4). 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 section 2.7.1 on page 2-17).

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    _f

LDR        A1, CON1
STR        A1, [sp, #0]
BL         _f

CON1:      .field -269488145,32
```

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.
- ☐ `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.

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.

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 changing the state of generated code (using the `.state16` or `.state32` directives)
- ☐ 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:

```
Label1: CMP    r1, #0        ; Compare r1 to zero.
        BCS    $1           ; If carry is set, branch to $1;
        ADDS   r0, r0, #1    ; else increment to r0
        MOVCS  pc, lr        ; and return.
$1:     LDR     r2, [r5], #4   ; Load indirect of r5 into r2
        ; with writeback.
        .newblock           ; Undefine $1 so it can be used
        ; again.
        ADDS   r1, r1, r2     ; Add r2 to r1.
        BPL    $1           ; If the negative bit isn't set,
        ; branch to $1;
        MVNS   r1, r1        ; else negate r1.
$1:     MOV     pc, lr        ; Return.
```

The following code uses a local label illegally:

```
Label1: CMP    r1, #0        ; Compare r1 to zero.
        BCS    $1           ; If carry is set, branch to $1;
        ADDS   r0, r0, #1    ; else increment to r0
        MOVCS  pc, lr        ; and return.
$1:     LDR     r2, [r5], #4   ; Load indirect of r5 into r2
        ; with writeback.
        ADDS   r1, r1, r2     ; Add r2 to r1.
        BPL    $1           ; If the negative bit isn't set,
        ; branch to $1;
        MVNS   r1, r1        ; else negate r1.
$1:     MOV     pc, lr        ; Return.
```

The `$1` label is not undefined before being reused by the last line of code. 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.

3.8.3 Symbolic Constants

You can set symbols 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
      .int    value            ;constant offsets value = 0
      .int    delta            ;constant offsets delta = 1
i_len  .endstruct

array  .tag     item            ;array declaration
      .bss    array, i_len*K
```

The assembler also has several predefined symbolic constants; these are discussed in section 3.8.5, *Predefined Symbolic Constants*, on page 3-23.

3.8.4 Defining Symbolic Constants (`-d` Option)

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

```
cl470 -dname=[value]
```

The *name* is the name of the symbol you want to define. The *value* is the value you want to assign to the symbol. If the *value* is omitted, the symbol is set to 1.

Once you have defined the name with the `-d` 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:

```
cl470 -dSYM1=1 -dSYM2=2 -dSYM3=3 -dSYM4=4 value.asm
```

Since you have assigned values to SYM1, SYM2, SYM3, and SYM4, you can use them in source code. Example 3-2 shows how the `value.asm` file uses these symbols without defining them explicitly.

Example 3–2. Using Symbolic Constants Defined on the 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 `-d` 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 <i>name</i> = <i>value</i></code>
Not equal to value	<code>.if <i>name</i> != <i>value</i></code>

The argument of 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** (the name of registers)
 - Coprocessor registers, including C0–C15
 - '470 registers, including R0–R15, and their aliases

The '470 register aliases are defined as follows:

Register Name	Alias	Register Name	Alias
R0	A1	R8	V5
R1	A2	R9	V6
R2	A3	R10	V7
R3	A4	R11	V8
R4	V1	R12	V9, IP
R5	V2	R13	SP
R6	V3	R14	LR
R7	V4, AP	R15	PC

Register symbols and aliases can be entered as all uppercase or all lowercase characters; that is, R13 could also be entered as r13, SP, or sp.

- ☐ **Coprocessor IDs**, including P0–P15. Coprocessor IDs are not case sensitive; that is, P12 could also be entered as p12.

- ❑ **Processor symbols**, as defined in the following table:

Symbol	Description
.TMS470	Always set to 1
.TMS470_16BIS	Set to 1 if the default state is 16 bit (the <code>-mt</code> assembler option is used); otherwise, set to 0
.TMS470_32BIS	Set to 1 if the default state is 32 bit (the <code>-mt</code> assembler option is not used); otherwise, set to 0
.TMS470_LITTLE	Set to 1 if little-endian mode is selected (the <code>-me</code> assembler option is used); otherwise, set to 0
.TMS470_BIG	Set to 1 if big-endian mode is selected (the <code>-me</code> assembler option is not used); otherwise, set to 0

Processor symbols can be entered as all uppercase or all lowercase characters; that is, `.TMS470` could also be entered as `.tms470`.

- ❑ **Status registers**, as defined in the following table:

Register	Alias	Description
CPSR	CPSR_ALL	Current processor status register
CPSR_FLG		Current processor status register flag bits only
SPSR	SPSR_ALL	Saved processor status register
SPSR_FLG		Saved processor status register flag bits only

Status registers can be entered as all uppercase or all lowercase characters; that is, `CPSR` could also be entered as `cpsr`, `CPSR_ALL`, or `cpsr_all`.

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:

```
.asg    "SP", stack-pointer
        ; Assigns the string SP to the substitution symbol
        ; stack-pointer.

.asg    "#0x20", block2
        ; Assigns the string #0x20 to the substitution
        ; symbol block2.

ADD     stack-pointer, stack-pointer, block2
        ; Adds the value in SP to #0x20 and stores the
        ; result in SP.
```

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:

```
addl    .macro dest, src
        ; addl macro definition

        ADDS    dest, dest, src
            ; Add the value in register dest to the value in
            ; register src, and store the result in src.
        BLCS    reset_ctr
            ; Handle overflow.
        .endm

*addl invocation
        addl R4, R5
            ; Calls the macro addl and substitutes R4 for dest
            ; and R5 for src. The macro adds the value of R4 and
            ; the value of R5, stores the result in R4, and
            ; handles overflow.
```

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 range of valid expression values is -2 147 483 648 to 4 294 967 295. Three main factors influence the order of expression evaluation:

Parentheses

Expressions enclosed in parentheses are always evaluated first.

$$8 / (4 / 2) = 4, \text{ but } 8 / 4 / 2 = 1$$

You *cannot* substitute braces ({ }) or brackets ([]) for parentheses.

Precedence groups

Operators, listed in Table 3-1, are divided into nine precedence groups. When parentheses do not determine the order of expression evaluation, the highest precedence operation is evaluated first.

$$8 + 4 / 2 = 10 \text{ (} 4 / 2 \text{ is evaluated first)}$$

Left-to-right evaluation

When parentheses and precedence groups do not determine the order of expression evaluation, operations are evaluated from left to right, except for Group 1, which is evaluated from right to left.

$$8 / 4 * 2 = 4, \text{ but } 8 / (4 * 2) = 1$$

3.9.1 Operators

Table 3–1 lists the operators that can be used in expressions, according to precedence group.

Table 3–1. Operators Used in Expressions (Precedence)

Group	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

Note: Group 1 operators are evaluated right to left. All other operators are evaluated left to right.

Note: Unary + and – have higher precedence than the binary forms.

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:

<code>=</code>	<code>[=]</code>	Equal to	<code>!=</code>	Not equal to
<code><</code>		Less than	<code><=</code>	Less than or equal to
<code>></code>		Greater than	<code>>=</code>	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 Relocatable Symbols and Legal Expressions

All legal expressions can be reduced to one of two forms:

relocatable symbol \pm *absolute symbol*

or

absolute value

Unary operators can be applied only to absolute values; they cannot be applied to relocatable symbols. Expressions that cannot be reduced to contain only one relocatable symbol are illegal.

Table 3–2 summarizes valid operations on absolute, relocatable, and external symbols. An expression cannot contain multiplication or division by a relocatable or external symbol. An expression cannot contain unresolved symbols that are relocatable to other sections.

Symbols that have been defined as global with the `.global` directive can also be used in expressions; in Table 3–2, these symbols are referred to as *external*.

Table 3–2. Expressions With Absolute and Relocatable Symbols

If A is...	and	If B is... , then	A + B is...	and	A – B is...
absolute		absolute	absolute		absolute
absolute		external	external		illegal
absolute		relocatable	relocatable		illegal
relocatable		absolute	relocatable		relocatable
relocatable		relocatable	illegal		absolute [†]
relocatable		external	illegal		illegal
external		absolute	external		external
external		relocatable	illegal		illegal
external		external	illegal		illegal

[†] A and B must be in the same section; otherwise, adding relocatable symbols to relocatable symbols is illegal.

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
LAB1:          .set 2               ; LAB1 = 2
intern_2       ; Relocatable, defined in current
                                   ; module
intern_3       ; Relocatable, defined in current
                                   ; module

```

❑ Example 1

The statements in this example use an absolute symbol, LAB1, which is defined to have a value of 2. The first statement loads the value 51 into R0. The second statement loads the value 27 into R0.

```

MOV  R0, #LAB1 + ((4+3) * 7) ; R0 = 51
                                ; 2 + ((7) * 7)
                                ; 2 + (49) = 51

MOV  R0, #LAB1 + 4 + (3*7) ; R0 = 27
                                ; 2 + 4 + (21) = 27

```

❑ Example 2

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

```
LDR R1, intern_1 - 10      ; Legal

LDR R1, 10-intern_1        ; Can't negate reloc. symbol
LDR R1, -(intern_1)        ; Can't negate reloc. symbol
LDR R1, intern_1/10        ; / isn't additive operator
LDR R1, intern_1 + intern_2 ; Multiple relocatables
```

❑ Example 3

The first statement in this example 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.

```
LDR R1, intern_1 - intern_2 + intern_3    ; Legal

LDR R1, intern_1 + intern_2 + intern_3    ; Illegal
```

❑ Example 4

A relocatable symbol's placement in the expression is important to expression evaluation. Although the statement here 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 `intern_3`.

```
LDR R1, intern_1 + intern_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 page 3-5).

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. Each field can contain one or more pieces of information, depending on the kind of statement it is associated with and the context of the program. Example 3-3 shows an assembler listing with each of the four fields identified. The following list describes the fields.

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 illustrated below:

!	undefined external reference
'	.text relocatable
"	.data relocatable
+	.sect relocatable
–	.bss, .usect relocatable

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.

Example 3–3 shows an assembler listing with each of the four fields identified.

3-34

3.11 Debugging Assembly Source

When you invoke `cl470` with `-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` 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 (see page 4-27) 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 page 4-44).

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

Example 3–4. Viewing Assembly Variables as C Types

(a) C Program `cvar.c`

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

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

(b) Assembly Program addfive.asm

```
-----  
; 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
```

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

```
cl470 -g cvars.c addfive.asm -z -l=lnk.cmd -l=rts32.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 page 3-5) or use the `.option` directive with the `X` operand (see page 4-60). The assembler appends the cross-reference to the end of the source listing. Example 3-5 shows a cross-reference listing; the text following describes the four fields in the listing.

Example 3-5. Assembler Cross-Reference Listing

LABEL	VALUE	-DEFN	REF			
.TMS470	00000001	0				
.TMS470_16BIS	00000000	0				
.TMS470_32BIS	00000001	0				
.TMS470_BIG	00000001	0				
.TMS470_LITTLE	00000000	0				
.tms470	00000001	0				
.tms470_16bis	00000000	0				
.tms470_32bis	00000001	0				
.tms470_big	00000001	0				
.tms470_little	00000000	0				
STACKSIZE	00000200	9	10	63		
__stack	00000000-	10	5	62		
dispatch	REF	29	60			
reset	00000000'	34	16	19	30	
stack	00000024'	62	52			
stacksz	00000028'	63	54			

Label	column contains each symbol that was defined or referenced during the assembly.
Value	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 can also be followed by a character that describes the symbol's attributes. Table 3-3 lists these characters and names.
Definition	(DEFN) column contains the statement number that defines the symbol. This column is blank for undefined symbols.
Reference	(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 3–3. 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

Assembler Directives

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

- ☐ 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 (sections 4.1 through 4.10) describes the directives according to function, and the second part (section 4.11) is an alphabetical reference.

Topic	Page
4.1 Assembler Directives Summary	4-2
4.2 Directives That Define Sections	4-7
4.3 Directives That Change the Instruction Type	4-10
4.4 Directives That Initialize Constants	4-11
4.5 Directives That Align the Section Program Counter	4-15
4.6 Directives That Format the Output Listing	4-17
4.7 Directives That Reference Other Files	4-19
4.8 Directives That Enable Conditional Assembly	4-20
4.9 Directives That Define Symbols at Assembly Time	4-21
4.10 Miscellaneous Directives	4-23
4.11 Directives Reference	4-24

4.1 Assembler Directives Summary

Table 4–1 summarizes the assembler directives.

Besides the assembler directives documented here, the TMS470R1x™ software tools support the following directives:

- ❑ The assembler uses several directives for macros. The macro directives are listed in this chapter, but they are described in detail in Chapter 5, *Macro Language*.
- ❑ The absolute lister also uses directives. Absolute listing directives are not entered by the user but are inserted into the source program by the absolute lister. Chapter 8, *Absolute Lister Description*, discusses these directives; they are not discussed in this chapter.
- ❑ 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.

Note: Labels and Comments Not Shown in Syntaxes

Any source statement that contains a directive may also contain a label and a comment. Labels begin in the first column (and they are the only elements, except comments, that 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 unless a label is required.

Table 4–1. Assembler Directives Summary

(a) *Directives that define sections*

Mnemonic and Syntax	Description	Page
.bss <i>symbol, size in bytes [, alignment]</i>	Reserves <i>size in bytes</i> in the .bss (uninitialized data) section	4-28
.click [<i>"section name"</i>]	Enables conditional linking for the current or specified section	4-31
.data	Assembles into the .data (initialized data) section	4-34
.sect " <i>section name</i> "	Assembles into a named (initialized) section	4-62
.text	Assembles into the .text (executable code) section	4-73
<i>symbol</i> .usect " <i>section name</i> ", <i>size in bytes</i> [, <i>alignment</i>]	Reserves <i>size in bytes</i> in a named (uninitialized) section	4-76

Table 4–1. Assembler Directives Summary (Continued)

(b) Directives that change the instruction type

Mnemonic and Syntax	Description	Page
.state16	Begins assembling 16-bit instructions	4-67
.state32	Begins assembling 32-bit instructions (default)	4-68

(c) Directives that initialize constants (data and memory)

Mnemonic and Syntax	Description	Page
.byte <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more successive bytes in the current section	4-30
.char <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more successive bytes in the current section	4-30
.double <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more 64-bit, IEEE double-precision, floating-point constants	4-35
.field <i>value</i> [, <i>size</i>]	Initializes a field of <i>size</i> bits (1–32) with <i>value</i>	4-41
.float <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more 32-bit, IEEE single-precision, floating-point constants	4-43
.half <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more 16-bit halfwords	4-46
.int <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more 32-bit integers	4-49
.long <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more 32-bit integers	4-49
.short <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more 16-bit halfwords	4-46
.string { <i>expr</i> ₁ "str ₁ " } [, ... , { <i>expr</i> _{<i>n</i>} "str _{<i>n</i>} " }]	Initializes one or more text strings	4-69
.word <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]	Initializes one or more 32-bit integers	4-49

(d) Directives that perform alignment and reserve space

Mnemonic and Syntax	Description	Page
.align [<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 word boundary (4 bytes)	4-24
.bes <i>size in bytes</i>	Reserves <i>size in bytes</i> in the current section; a label points to the last addressable byte in the reserved space	4-64
.space <i>size in bytes</i>	Reserves <i>size in bytes</i> in the current section; a label points to the beginning of the reserved space	4-64

Table 4–1. Assembler Directives Summary (Continued)

(e) Directives that format the output listing

Mnemonic and Syntax	Description	Page
.drlist	Enables listing of all directive lines (default)	4-36
.drnolist	Suppresses listing of certain directive lines	4-36
.fclist	Allows false conditional code block listing (default)	4-40
.fcnolist	Suppresses false conditional code block listing	4-40
.length <i>page length</i>	Sets the page length of the source listing	4-51
.list	Restarts the source listing	4-52
.mlist	Allows macro listings and loop blocks (default)	4-57
.mnolist	Suppresses macro listings and loop blocks	4-57
.nolist	Suppresses the source listing	4-52
.option <i>option₁</i> [, <i>option₂</i> , . . .]	Selects output listing options; available options are A, B, H, M, N, O, R, T, W, and X	4-60
.page	Ejects a page in the source listing	4-62
.sslist	Allows expanded substitution symbol listing	4-66
.ssnolist	Suppresses expanded substitution symbol listing (default)	4-66
.tab <i>size</i>	Sets tab size to <i>size</i> characters	4-72
.title " <i>string</i> "	Prints a title in the listing page heading	4-74
.width <i>page width</i>	Sets the page width of the source listing to <i>page width</i>	4-51

Table 4–1. Assembler Directives Summary (Continued)

(f) Directives that reference other files

Mnemonic and Syntax	Description	Page
.copy [""]filename[""]	Includes source statements from another file	4-32
.def symbol ₁ [, ... , symbol _n]	Identifies one or more symbols that are defined in the current module and that can be used in other modules	4-44
.global symbol ₁ [, ... , symbol _n]	Identifies one or more global (external) symbols	4-44
.include [""]filename[""]	Includes source statements from another file	4-32
.ref symbol ₁ [, ... , symbol _n]	Identifies one or more symbols used in the current module that are defined in another module	4-44

(g) Directives that define symbols

Mnemonic and Syntax	Description	Page
.asg [""]character string[""], substitution symbol	Assigns a character string to <i>substitution symbol</i>	4-25
symbol .equ value	Equates <i>value</i> with <i>symbol</i>	4-63
.eval well-defined expression, substitution symbol	Performs arithmetic on numeric <i>substitution symbol</i>	4-25
.label symbol	Defines a load-time relocatable label in a section	4-50
symbol .set value	Equates <i>value</i> with <i>symbol</i>	4-63

(h) Directives that define macros

Mnemonic and Syntax	Description	Page
.macro	Identify the source statement as the first line of a macro definition. You must place .macro in the opcode field	4-55
.mlib [""]filename[""]	Define macro library	4-55
.mexit	Go to .endm . This directive is useful when error testing confirms that macro expansion will fail.	5-3
.endm	End .macro code block	4-39
.var	Define a local macro substitution symbol	4-78

Table 4–1. Assembler Directives Summary (Continued)

(i) Directives that control conditional assembly

Mnemonic and Syntax	Description	Page
.break [<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.	4-53
.else	Assembles code block if the .if <i>well-defined expression</i> is false. When using the .if construct, the .else construct is optional.	4-47
.elseif <i>well-defined expression</i>	Assembles code block if the .if <i>well-defined expression</i> is false and the .elseif condition is true. When using the .if construct, the .elseif construct is optional.	4-47
.endif	Ends .if code block	4-47
.endloop	Ends .loop code block	4-53
.if <i>well-defined expression</i>	Assembles code block if the <i>well-defined expression</i> is true	4-47
.loop [<i>well-defined expression</i>]	Begins repeatable assembly of a code block; the loop count is determined by the <i>well-defined expression</i> .	4-53

(j) Directives that define structures

Mnemonic and Syntax	Description	Page
.endstruct	Ends structure definition	4-70
.struct	Begins structure definition	4-70
.tag	Assigns structure attributes to a label	4-70

(k) Miscellaneous directives

Mnemonic and Syntax	Description	Page
.asmfunc	Identify the beginning of a block of code that contains a function	
.emsg <i>string</i>	Sends user-defined error messages to the output device; produces no .obj file	4-37
.end	Ends program	4-39
.endasmfunc	Identify the end of a block of code that contains a function	
.mmsg <i>string</i>	Sends user-defined messages to the output device	4-37
.newblock	Undefines local labels	4-59
.wmsg <i>string</i>	Sends user-defined warning messages to the output device	4-37

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 **.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 **.data** directive identifies portions of code in the .data section. The .data section usually contains initialized data.
- ☐ The **.sect** directive defines initialized named sections 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:

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

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
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 Change the Instruction Type

By default, the assembler begins assembling all instructions in a file as 32-bit instructions. You can change the default action by using the `-mt` assembler (see page 3-6) option, which causes the assembler to begin assembling all instructions in a file as 16-bit instructions. You can also use two directives that change how the assembler assembles instructions starting at the point where the directives occur:

- ❑ The **.state16** directive causes the assembler to begin assembling 16-bit instructions starting at the location of the directive. The `.state16` directive performs an implicit halfword alignment before any instructions are written to the section to ensure that all 16-bit instructions are halfword aligned. The `.state16` directive also resets any local labels defined.
- ❑ The **.state32** directive tells the assembler to begin assembling 32-bit instructions starting at the location of the directive. The `.state32` directive performs an implicit word alignment before any instructions are written to the section to ensure that all 32-bit instructions are word aligned. The `.state32` directive also resets any local labels defined.

4.4 Directives That Initialize Constants

Several directives assemble values for the current section:

- ❑ The **.bes** and **.space** directives reserve a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s.
You can reserve a specified number of words by multiplying the number of bytes by 4.
- 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 4–1 shows how the **.space** and **.bes** directives work for the following assembled code:

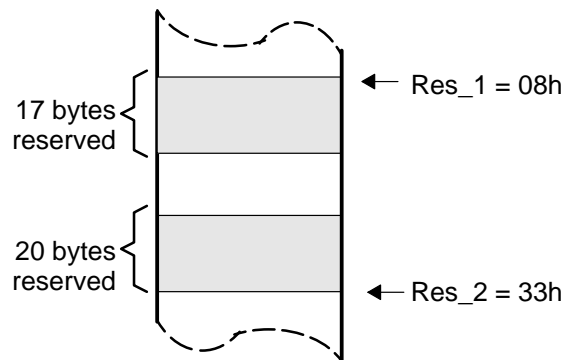
```

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 BA                .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 4–1. The **.space** and **.bes** Directives



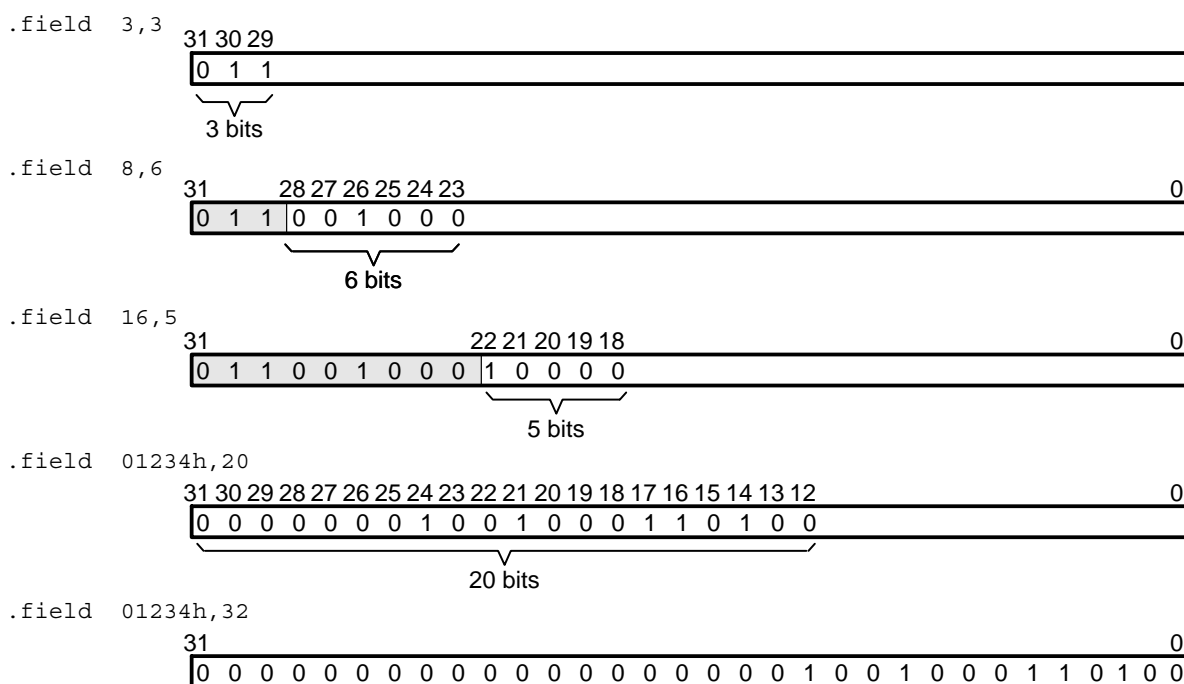
- ❑ The **.byte** and **.char** directives place one or more 8-bit values into consecutive bytes of the current section. These directives are similar to **.word** (described on page 4-13), except that the width of each value is restricted to eight bits.
- ❑ The **.double** directive calculates the double-precision (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 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 4–2 shows how fields are packed into a word using the following assembled code; notice that the SPC does not change for the first three fields (the fields are packed into the same word).

1	00000000	60000000	.field	3, 3
2	00000000	64000000	.field	8, 6
3	00000000	64400000	.field	16, 5
4	00000004	01234000	.field	01234h, 20
5	00000008	00001234	.field	01234h, 32

Figure 4–2. The **.field** Directive



Note: This figure uses big-endian configuration.

- ❑ The **.float** directive calculates the single-precision (32-bit) IEEE floating-point representation of one or more floating-point values and stores them in the current section. The **.float** directive automatically aligns to a word boundary.
- ❑ The **.half** and **.short** directives place one or more 16-bit halfword values into consecutive halfwords in the current section.

- ❑ The **.int**, **.long**, and **.word** directives place one or more 32-bit values into consecutive words in the current section.
- ❑ The **.string** directive places 8-bit characters from one or more character strings into the current section. The **.string** directive is similar to **.byte**, placing an 8-bit character in each consecutive word of the current section.


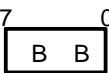
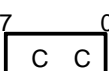
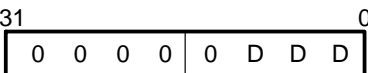

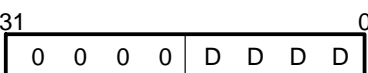

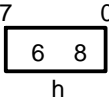
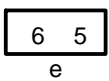
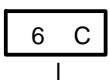
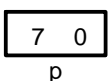
Note: The **.byte**, **.char**, **.word**, **.int**, **.long**, **.string**, **.double**, **.float**, **.half**, **.short**, and **.field** Directives in a **.struct/.endstruct** Sequence

The **.byte**, **.char**, **.word**, **.int**, **.long**, **.string**, **.double**, **.float**, **.half**, **.short**, 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 about the **.struct/.endstruct** directives, see page 4-70.

Figure 4–3 compares the **.byte**, **.char**, **.int**, **.long**, **.float**, **.word**, and **.string** directives, using the following assembled code:

1	00000000	AA	.byte	0AAh, 0BBh
	00000001	BB		
2	00000002	CC	.char	0CCh
3	00000004	00000DDD	.word	0DDdh
4	00000008	EEEEFFFF	.long	0EEEEFFFFh
5	0000000c	00000DDD	.int	0DDdh
6	00000010	3FFFFFFAC	.float	1.99999
7	00000014	68	.string	"help"
	00000015	65		
	00000016	6C		
	00000017	70		
8				

Figure 4–3. Initialization Directives

Byte		Code
0		<code>.byte 0AAh</code>
1		<code>.byte 0BBh</code>
2		<code>.char 0CCh</code>
4		<code>.word 0DDDh</code>
8		<code>.long EEEEEFFFh</code>
c		<code>.int 0DDDDh</code>
10		<code>.float 1.99999</code>
14		<code>.string "help"</code>
15		
16		
17		

4.5 Directive That Aligns the Section Program Counter

The **.align** directive aligns the SPC at a 1-byte to 32K-byte boundary. This ensures that the code following the directive begins on the byte value that you specify. If the SPC is already aligned at the selected boundary, it is not incremented. Operands for the **.align** directive must equal a power of 2 between 2^0 and 2^{15} , inclusive. For example:

Operand of 2 aligns SPC to halfword boundary
 4 aligns SPC to word boundary
 128 aligns SPC to 128-byte page boundary

The **.align** directive with no operands defaults to 4, that is, to a word boundary.

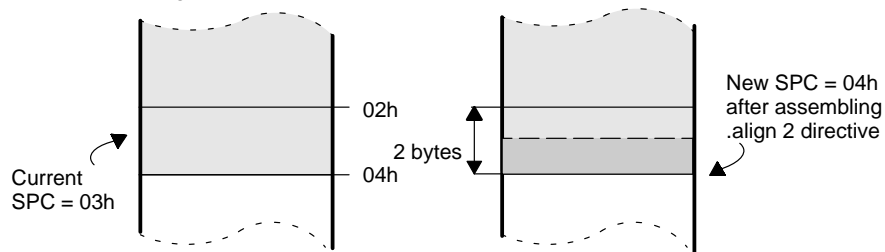
Figure 4–4 demonstrates the **.align** directive, using the following assembled code:

```

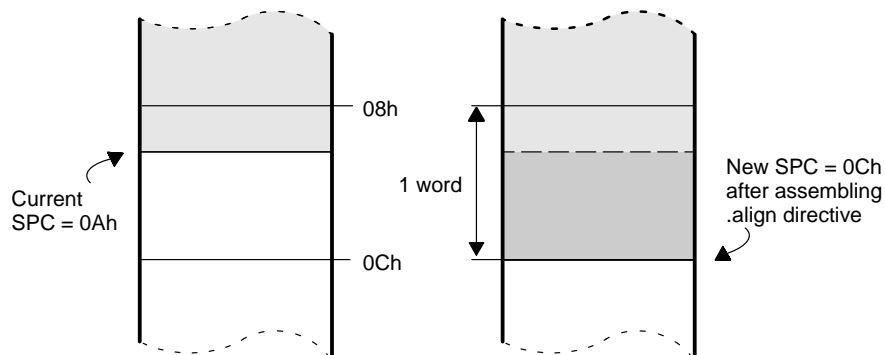
1 00000000 40000000      .field      2,3
2 00000000 4000000B      .field      11, 21
3                          .align      2
4 00000004 45            .string      "Errcnt"
   00000005 72
   00000006 72
   00000007 63
   00000008 6E
   00000009 74
5                          .align
6 0000000c 04            .byte      4
```

Figure 4-4. The `.align` Directive

(a) Result of `.align 2`



(b) Result of `.align` without an argument



4.6 Directives That Format the Output Listing

The following 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:

.asg	.eval,	.length	.mnolist	.var
.break	.fclist	.mlist	.sslist	.width
.emsg	.fcnolist	.mmsg	.ssnolist	.wmsg

You can use the **.drlist** directive to turn the listing on again.

- ☐ 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.
 - H** limits the listing of **.half** and **.short** 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**, **M**, **T**, and **W** options (turns off the limits of **B**, **H**, **M**, **T**, and **W**).
 - T** limits the listing of **.string** directives to one line.

- W** limits the listing of `.int`, `.long`, and `.word` 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.
- ☐ 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.

4.7 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 by 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*, on page 2-17.) 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 **.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 that the linker can resolve its definition.

4.8 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 assemble a block of code conditionally according to the evaluation of an expression.

.if *well-defined expression* marks the beginning of a conditional block and assembles code if the *.if well-defined expression* is true.

[.elseif *well-defined expression*] marks a block of code to be assembled if the *.if well-defined expression* is false and the *.elseif* condition is true.

[.else] marks a block of code to be assembled if the *.if well-defined expression* is false.

[.endif] marks the end of a conditional block and terminates the block.

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

.loop [*well-defined expression*] marks the beginning a repeatable block of code. The optional expression evaluates to the loop count.

[.break [*well-defined expression*]] tells the assembler to assemble repeatedly when the *.break well-defined expression* is false and to go to the code immediately after *.endloop* when the expression is true or omitted.

.endloop 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*, on page 3-28.

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
        ; Assign string to substitution symbol.

.byte   coefficients
        ; Place the symbol values 10, 20, 30, and 40
        ; into consecutive bytes in current section.
```

- ❑ 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      ; x = 1
.loop     ; Begin conditional loop.
.byte     x*10h      ; Store value into current section.
.break    x = 4      ; Break loop if x = 4.
.eval     x+1, x      ; Increment x by 1.
.endloop   ; End conditional loop.
```

- ❑ The **.label** directive defines a special symbol that refers to the loadtime 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 page 4-50 for an example using a loadtime 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    0100h
        ; Set bval = 0100h
        .long    bval, bval*2, bval+12
        ; Store the values 0100h, 0200h, and 010Ch
        ; into consecutive words in current section.
```

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

- ❑ The **.struct/.endstruct** directives set up C-like structure definitions, and the **.tag** directive assigns the C-like structure characteristics to a label.

The **.struct/.endstruct** directives allow you to organize your information into structures so that similar elements can be grouped together. Element offset calculation is left up to the assembler. The **.struct/.endstruct** 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. 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.

```
type  .struct           ; structure tag definition
X     .int
Y     .int
T_LEN .endstruct

COORD .tag type         ; declare COORD (coordinate)

COORD .space T_LEN      ; actual memory allocation

LDR   R0, COORD.Y       ; load member Y of structure
                        ; COORD into register R0.
```


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 **.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. They are defined when they appear in the label field. Local labels are temporary labels that can be used as operands for branch 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*, on page 3-19.

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*, on page 5-17.

4.11 Directives Reference

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

.align

Align SPC on a 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 may be any power of 2 between 2⁰ and 2¹⁵, inclusive. An operand of 4 aligns the SPC on the next word boundary, and this is the default if no size is given. The assembler assembles NOP instructions up to the next boundary that you specify. For example:

Operand of 2 aligns SPC to halfword boundary
 4 aligns SPC to word boundary
 128 aligns SPC to 128-byte boundary

Using the `.align` directive has two effects:

- ☐ The assembler aligns the SPC on a byte value that you specify *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 4`, and a default `.align`.

```
1 00000000 04          .byte      4
2                          .align    2
3 00000002 45          .string    "Errorcnt"
  00000003 72
  00000004 72
  00000005 6F
  00000006 72
  00000007 63
  00000008 6E
  00000009 74
4
5 0000000c 60000000     .align
6 0000000c 6A000000     .field    3,3
7                          .field    5,4
8 0000000c 6A006000     .align    2
9                          .field    3,3
10 00000010 50000000    .align    8
11                          .field    5,4
12 00000014 04          .align
                          .byte      4
```

.asg/.eval*Assign a Substitution Symbol*

Syntax

.asg ["]*character string*["], *substitution symbol*
.eval *well-defined expression*, *substitution symbol*

Description

The **.asg** directive assigns a character string to a substitution symbol. 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* is a required parameter that must be a valid symbol name. The substitution symbol is up to 32 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* is a required parameter that must be a valid symbol name. The substitution symbol is up to 32 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 sub. symbols
2                                ; using .asg and .eval
3
4                                .asg    R13, STACKPTR
5                                .asg    &, AND
6
7 00000000 E28DD018            ADD     STACKPTR, STACKPTR, #280 AND 255
#                               ADD     R13, R13, #280 & 255
8 00000004 E28DD018            ADD     STACKPTR, STACKPTR, #280 & 255
#                               ADD     R13, R13, #280 & 255
9
10                               .asg    0, x
11                               .loop    5
12                               .eval    x+1, x
13                               .word    x
14                               .endloop
1                               .eval    x+1, x
#                               .eval    0+1, x
1                               .word    x
#                               .word    1
1                               .eval    x+1, x
#                               .eval    1+1, x
1 0000000c 00000002            .word    x
#                               .word    2
1                               .eval    x+1, x
#                               .eval    2+1, x
1 00000010 00000003            .word    x
#                               .word    3
1                               .eval    x+1, x
#                               .eval    3+1, x
1 00000014 00000004            .word    x
#                               .word    4
1                               .eval    x+1, x
#                               .eval    4+1, x
1 00000018 00000005            .word    x
#                               .word    5
```

**.asmfunc/
.endasmfunc***Mark Function Boundaries*

Syntax

symbol **.asmfunc**
.endasmfunc

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                                           .global userfunc
3                                           .global _printf
4
5                                userfunc: .asmfunc
6 00000000 00000010!                   CALL    .S1      _printf
7 00000004 01BC94F6                   STW     .D2T2   B3,*B15--(16)
8 00000008 01800E2A'                   MVKL    .S2      RL0,B3
9 0000000c 01800028+                   MVKL    .S1      SL1+0,A3
10 00000010 01800068+                  MVKH     .S1      SL1+0,A3
11
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*]

Description

The **.bss** directive reserves space for variables in the `.bss` section. This directive is usually used to allocate variables 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 for which you are reserving space.
- ☐ The *size in bytes* is a required parameter; it must be an absolute expression. The assembler allocates size bytes in the `.bss` section. There is no default size.

- ❑ The *alignment* is an optional parameter. It ensures that the space allocated to the symbol occurs on the specified boundary. The boundary indicates the size of the alignment in bytes and must be set to a power of 2 between 2^0 and 2^{15} , inclusive. If the SPC is aligned at the specified boundary, it is not incremented.

For more information about COFF sections, see Chapter 2, *Introduction to Common Object File Format*.

Example

In this example, the `.bss` directive allocates space for two variables, `TEMP` and `ARRAY`. The symbol `TEMP` points to four bytes of uninitialized space (at `.bss` `SPC = 0`). The symbol `ARRAY` points to 100 bytes of uninitialized space (at `.bss` `SPC = 04h`). Symbols declared with the `.bss` directive can be referenced in the same manner as other symbols and can also be declared external.

```
1          *****
2          ** Start assembling into the .text section. **
3          *****
4 00000000      .text
5 00000000 E3A00000      MOV      R0, #0
6
7          *****
8          ** Allocate 4 bytes in .bss for TEMP.      **
9          *****
10 00000000      Var_1: .bss      TEMP, 4
11
12          *****
13          ** Still in .text.                          **
14          *****
15 00000004 E2801056      ADD      R1, R0, #56h
16 00000008 E0020091      MUL      R2, R1, R0
17
18          *****
19          ** Allocate 100 bytes in .bss for the symbol **
20          ** named ARRAY.                          **
21          *****
22 00000004          .bss      ARRAY, 100, 4
23
24          *****
25          ** Assemble more code into .text.          **
26          *****
27 0000000c E1A0F00E      MOV      PC, LR
28
29          *****
30          ** Declare external .bss symbols.          **
31          *****
32          .global ARRAY, TEMP
33          .end
```

.byte/.char	<i>Initialize Byte</i>
Syntax	.byte <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}] .char <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}]
Description	<p>The .byte and .char directives place one or more values into consecutive bytes of the current section. A <i>value</i> can be either:</p> <ul style="list-style-type: none"><input type="checkbox"/> An expression that the assembler evaluates and treats as an 8-bit signed number<input type="checkbox"/> A character string enclosed in double quotes. Each character in a string represents a separate value. <p>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.</p> <p>If you use a label, it points to the location where the assembler places the first byte.</p> <p>Note that 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 about .struct/.endstruct, see page 4-70.</p>
Example	<p>In this example 8-bit values (10, -1, abc, and a) are placed into consecutive bytes in memory with .byte. Also, 8-bit values (8, -3, def, and b) are placed into consecutive bytes in memory with .char. The label STRX has the value 100h, which is the location of the first initialized word.</p>

```
1 00000000                                .space 100h
2 00000100 0A                            STRX  .byte 10, -1, "abc", 'a'
   00000101 FF
   00000102 61
   00000103 62
   00000104 63
   00000105 61
3 00000106 08                                .char 8, -3, "def", 'b'
   00000107 FD
   00000108 64
   00000109 65
   0000010a 66
   0000010b 62
```


.clink*Conditionally Leave Section Out of COFF Output***Syntax****.clink** ["*section name*"]**Description**

The **.clink** directive sets up conditional linking for a section. 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 program is defined should not be marked as a conditionally linked section.

Example

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

```

1          *****
2          ** Set Vars section for conditional linking.  **
3          *****
4 00000000          .sect "Vars"
5                  .clink
6 00000000 000000AA X:      .word 0AAh
7 00000004 000000AA Y:      .word 0AAh
8 00000008 000000AA Z:      .word 0AAh
9          *****
10         ** Set Counts section for conditional linking. **
11         *****
12 00000000          .sect "Counts"
13                 .clink
14 00000000 000000AA XCount: .word 0AAh
15 00000004 000000AA YCount: .word 0AAh
16 00000008 000000AA ZCount: .word 0AAh
17         *****
18         ** .text is unconditionally linked by default. **
19         *****
20 00000000          .text
21 00000000 E59F0004    LDR  R0, X_addr
22 00000004 E5900000    LDR  R0, [R0]
23 00000008 E0800001    ADD  R0, R0, R1
24
25 0000000c 00000000+ X_addr: .field X, 32
26         *****
27         ** The reference to symbol X causes the Vars    **
28         ** section to be linked into the COFF output.  **
29         *****

```

.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, c:\470 tools\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 **A_DIR** environment variable .

For more information about the **-i** option and **A_DIR**, see section 3.4, *Naming Alternate Directories for Assembler Input*, on page 3-7.

The **.copy** and **.include** directives 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.

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 (see the following listing file). 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>

Listing file:

```

1 00000000          .space 29
2          .copy "byte.asm"
A 1          ** In byte.asm
A 2 0000001d 20      .byte 32,1+ 'A'
      0000001e 42
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 6A      .byte 67h + 3q
      3
      4          ** Back in original file
      5 00000029 64      .string "done"
      0000002a 6F
      0000002b 6E
      0000002c 65
```

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.

include.asm (source file)	byte2.asm (first include file)	word2.asm (second include file)
<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 64      .string "done"
   0000002a 6F
   0000002b 6E
   0000002c 65
```

.data

Assemble Into .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.

The assembler assumes that `.text` is the default section. If you want the assembler to assemble code into the `.data` section at the beginning of assembly, you must use the `.data` directive to instruct it to do so.

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                ; Constant into .data
11          00000000 INDEX .set 0
12 00000000 E3A00000 MOV R0, #INDEX
13
14          *****
15          **                Assemble into .data.                **
16          *****
17 000000cc          Table: .data
18 000000cc FFFFFFFF          .word -1          ; Assemble 32-bit
19                                     ; constant into .data.
20
21 000000d0 FF          .byte 0FFh          ; Assemble 8-bit
22                                     ; constant into .data.
23
24          *****
25          **                Assemble into .text.                **
26          *****
27 00000004          .text
28 00000004 000000CC" con: .field Table, 32
29 00000008 E51F100C LDR R1, con
30 0000000c E5912000 LDR R2, [R1]
31 00000010 E0802002 ADD R2, R0, R2
32          *****
33          ** Resume assembling into the .data section **
34          ** at address 0Fh. **
35          *****
36 000000d1          .data
```

.double

Initialize Double-Precision Floating-Point Value

Syntax

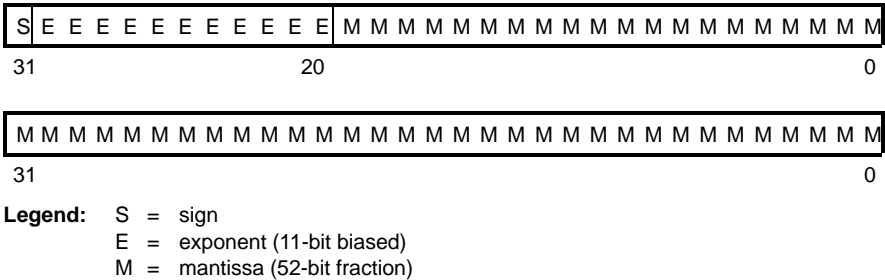
.double *value*₁ [, ... , *value*_{*n*}]

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. Floating-point constants are aligned on a 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 about `.struct/.endstruct`, see page 4-70.

Example

This example shows the `.double` directive.

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

.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.

- | | | |
|---|---|---|
| <input type="checkbox"/> <code>.asg</code> | <input type="checkbox"/> <code>.fcnolist</code> | <input type="checkbox"/> <code>.ssnolist</code> |
| <input type="checkbox"/> <code>.break</code> | <input type="checkbox"/> <code>.mlist</code> | <input type="checkbox"/> <code>.var</code> |
| <input type="checkbox"/> <code>.emsg</code> | <input type="checkbox"/> <code>.mmsg</code> | <input type="checkbox"/> <code>.wmsg</code> |
| <input type="checkbox"/> <code>.eval</code> | <input type="checkbox"/> <code>.mnolist</code> | |
| <input type="checkbox"/> <code>.fclist</code> | <input type="checkbox"/> <code>.sslist</code> | |

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:

```
.asg      0, x
.loop     2
.eval     x+1, x
.endloop

.drnolist

.asg      1, x
.loop     3
.eval     x+1, x
.endloop
```

Listing file:

```

1          .asg      0, x
2          .loop     2
3          .eval     x+1, x
4          .endloop
1          .eval     0+1, x
1          .eval     1+1, x
          5
          6          .drnolist
          7
          9          .loop     3
10         .eval     x+1, x
11         .endloop
```

**.emsg/.mmsg/
.wmsg***Define Messages*

Syntax

```
.emsg string
.mmsg string
.wmsg string
```

Description

These directives allow you to define your own error and warning messages. When you use these directives, the assembler tracks the number of errors and warnings it encounters and prints these numbers on the last line of the listing file.

The **.emsg** directive sends an error message to the standard output device in the same manner as the assembler, incrementing the error count and preventing the assembler from producing an object file.

The **.mmsg** directive sends an assembly-time message to the standard output device in the same manner as the `.emsg` and `.wmsg` directives; however it does not set the error or warning counts, and it does not prevent the assembler from producing an object file.

The **.wmsg** directive sends a warning message to the standard output device in the same manner as the **.emsg** directive; however it increments the warning count rather than the error count, and it does not prevent the assembler from producing an object file.

Example

In this example, the message **ERROR -- MISSING PARAMETER** is sent to the standard output device.

Source file:

```
MSG_EX .macro parm1
        .if    $$symlen(parm1) = 0
        .emsg  "ERROR -- MISSING PARAMETER"
        .else
        add    parm1, r7, r8
        .endif
        .endm

MSG_EX R0

MSG_EX
```

Listing file:

```

1          MSG_EX .macro parm1
2          .if    $$symlen(parm1) = 0
3          .emsg  "ERROR -- MISSING PARAMETER"
4          .else
5          add    parm1, r7, r8
6          .endif
7          .endm
8
9 00000000          MSG_EX R0
1          .if    $$symlen(parm1) = 0
1          .emsg  "ERROR -- MISSING PARAMETER"
1          .else
1          add    R0, r7, r8
1          .endif
10
11 00000004          MSG_EX
1          .if    $$symlen(parm1) = 0
1          .emsg  "ERROR -- MISSING PARAMETER"
1 ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
1          .else
1          add    parm1, r7, r8
1          .endif

1 Error, No Warnings
```

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

```
*** ERROR!   line 11: ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
               .emsg  "ERROR -- MISSING PARAMETER"

1 Error, No 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 should 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.

Note: Ending a Macro

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:  .space  300
TEMP    .set    15
        .bss    LOC1, 48h
LOCL_n  .word    LOC1
        MVN     R0, R0
        ADD     R0, R0, #TEMP
        LDR     R4, LOCL_n
        STR     R0, [R4]
        .end
        .byte   4
        .word   CCCh
```

Listing file:

```
1 00000000          START:  .space  300
2          0000000F TEMP    .set    15
3 00000000          .bss    LOC1, 48h
4 0000012c 00000000- LOCL_n  .word    LOC1
5 00000130 E1E00000      MVN     R0, R0
6 00000134 E280000F      ADD     R0, R0, #TEMP
7 00000138 E51F4014      LDR     R4, LOCL_n
8 0000013c E5840000      STR     R0, [R4]
9                      .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:

```
AAA    .set  1
BBB    .set  0
      .fclist
      .if   AAA
      ADD   R0, R0, #1024
      .else
      ADD   R0, R0, #1024*10
      .endif
      .fcnolist
      .if   AAA
      ADD   R0, R0, #1024
      .else
      ADD   R0, R0, #1024*10
      .endif
```

Listing file:

```
1          00000001  AAA    .set  1
2          00000000  BBB    .set  0
3                                     .fclist
4
5                                     .if   AAA
6 00000000  E2800B01  ADD   R0, R0, #1024
7                                     .else
8                                     ADD   R0, R0, #1024*10
9                                     .endif
10
11                                     .fcnolist
12
13 00000004  E2800B01  ADD   R0, R0, #1024
```

.field*Initialize Field*

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

The **.field** directive can initialize multiple-bit fields within a single word (32 bits) 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. If the value is relocatable, *size in bits* must be 32.
- ❑ 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 that the size is 32 bits. If you specify a value that cannot fit into the given size, the assembler truncates the value and issues an error message. For example, **.field 3,1** causes the assembler to truncate the value 3 to 1; the assembler also prints the message:

```
*** WARNING! line 11: W0001: Field value truncated to 1
      .field 3,1
```

Successive **.field** directives pack values into the specified number of bits starting at the current word. Fields are packed starting at the most significant part of the word, moving toward the least significant part as more fields are added. If the assembler encounters a field size that does not fit into the current word, it writes out the word, and begins packing fields into the next word.

You can use the **.align** directive with an operand of 4 to force the next **.field** directive to begin packing into a new word.

If you use a label with **.field**, it points to the word that contains the specified field, not the starting address of the specific field.

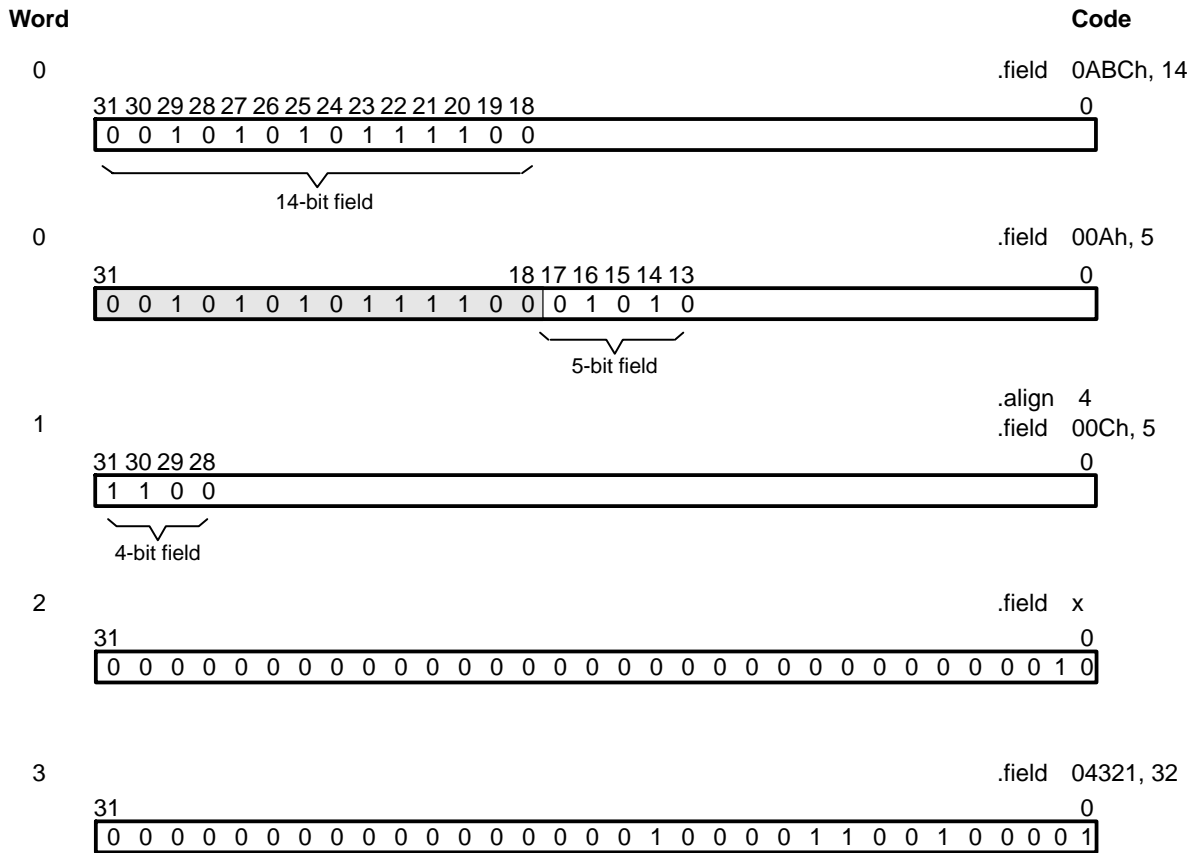
When you use **.field** in a **.struct/.endstruct** sequence, **.field** defines a member's size; it does not initialize memory. For more information about **.struct/.endstruct**, see page 4-70.

Example

This example shows how fields are packed into a word. Notice that the SPC does not change until a word is filled and the next word is begun. For more examples of the .field directive, see page 4-12.

```
1 *****
2 **      Initialize a 14-bit field.  **
3 *****
4 00000000 2AF00000      .field 0ABCh, 14
5
6 *****
7 **      Initialize a 5-bit field      **
8 **      in the same word.            **
9 *****
10 00000000 2AF14000 L_F: .field 0Ah, 5
11
12 *****
13 **      Write out the word.          **
14 *****
15 .align 4
16
17 *****
18 **      Initialize a 4-bit field.      **
19 ** This fields starts a new word. **
20 *****
21 00000004 C0000000 x:   .field 0Ch, 4
22
23 *****
24 **      32-bit relocatable field      **
25 **      in the next word.            **
26 *****
27 00000008 00000004' .field x
28
29 *****
30 **      Initialize a 32-bit field.      **
31 *****
32 0000000c 00004321 .field 04321h, 32
```

Figure 4-6 shows how the directives in this example affect memory. This example assumes big-endian format.

Figure 4–6. The .field Directive**.float***Initialize Single-Precision Floating-Point Value*

Syntax**.float** *value*₁ [, ... , *value*_{*n*}]**Description**

The **.float** directive places the IEEE single-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 single-precision 32-bit format. Floating-point constants are aligned on word boundaries.

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 also 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 E2800056      ADD      R0, R0, #56h
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 E2800056      ADD      R0, R0, #56h
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*Initialize 16-Bit Integers*

Syntax

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

Description

The **.half** and **.short** directives place one or more values into consecutive halfwords of the current section. A *value* can be either:

- ☐ An expression that the assembler evaluates and treats as a 16-bit signed 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 up to 100 value parameters, 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.

When you use `.half` or `.short` in a `.struct/.endstruct` sequence, `.half` and `.short` define a member's size; they do not initialize memory. For more information about `.struct/.endstruct`, see page 4-70.

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 100h, which is the location of the first initialized halfword for `.short`.

```
1 00000000                                .space 100h * 16
2 00001000 000A                            .half 10, -1, "abc", 'a'
   00001002 FFFF
   00001004 0061
   00001006 0062
   00001008 0063
   0000100a 0061
3 0000100c 0008                            STRN .short 8, -3, "def", 'b'
   0000100e FFFD
   00001010 0064
   00001012 0065
   00001014 0066
   00001016 0062
```

.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*, on page 3-28.

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 04      .byte    SYM4          ; Equal values
8              .else
9              .byte    SYM2 * SYM2      ; Unequal values
10             .endif
11
12          If_5:    .if      SYM1 <= 10
13 00000001 0A      .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 08      .byte    SYM4 + SYM4      ; Equal values
22             .endif
23
24          If_7:    .if      SYM1 = SYM2
25              .byte    SYM1
26              .elseif    SYM2 + SYM3 = 5
27 00000003 05      .byte    SYM2 + SYM3
28             .endif
```

.int/.long/.word*Initialize 32-Bit Integer*

Syntax

```
.int value1 [, ... , valuen]  
.long value1 [, ... , valuen]  
.word value1 [, ... , valuen]
```

Description

The **.int**, **.long**, and **.word** directives place one or more values into consecutive words in the current section. 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.

The **.int**, **.long**, and **.word** directives perform a word (32-bit) alignment before any data is written to the section.

When you use **.int**, **.long**, or **.word** in a **.struct/.endstruct** sequence, they define a member's size; they do not initialize memory. For more information about **.struct/.endstruct**, see page 4-70.

Example 1

In this example, the **.int** directive is used to initialize words.

```
1 00000000 .space 73h  
2 00000000 .bss PAGE, 128  
3 00000080 .bss SYMPTR, 4  
4 00000074 E3A00056 INST: MOV R0, #056h  
5 00000078 0000000A .int 10, SYMPTR, -1, 35 + 'a', INST, "abc"  
0000007c 00000080-  
00000080 FFFFFFFF  
00000084 00000084  
00000088 00000074'  
0000008c 00000061  
00000090 00000062  
00000094 00000063
```

Example 2 In this example, the `.word` directive is used to initialize words. The symbol `WORDX` points to the first word that is reserved.

```
1 00000000 00000C80 WORDX: .word 3200, 1 + 'AB', -0AFh, 'X'
   00000004 00004242
   00000008 FFFFFFF51
   0000000c 00000058
```

Example 3 This example shows how the `.long` directive initializes words. The symbol `DAT1` points to the first word that is reserved.

```
1 00000000 0000ABCD DAT1: .long 0ABCDh, 'A' + 100h, 'g', 'o'
   00000004 00000141
   00000008 00000067
   0000000c 0000006F
2 00000010 00000000' .long DAT1, 0AABBCCDDh
   00000014 AABBCDD
3 00000018 DAT2:
```

.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 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 details about assigning run-time and load-time addresses in the linker, see section 7.9, *Specifying a Section's Run-Time Address*, on page 7-44.

.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: 80 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 80 characters.
- ☐ Minimum width: 80 characters
- ☐ Maximum width: 512 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 allow 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 specified. However, if you do not request a listing file when you invoke the assembler (by including the **-l** option on the command line), the assembler ignores the **.list** directive.

Example

This example shows how the `.copy` directive inserts source statements from another file. The first time this directive is encountered, the assembler lists the copied source lines in the listing file. The second time this directive is encountered, the assembler does not list the copied source lines, because a `.nolist` directive was assembled. Note that the `.nolist`, the second `.copy`, 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:

```
.copy "copy2.asm"
* Back in original file
NOP
.nolist
.copy "copy2.asm"
.list
* Back in original file
.string "Done"
```

Listing file:

```

1                                     .copy  "copy2.asm"
A   1                                     * In copy2.asm (copy file)
A   2 00000000 00000020             .word 32, 1 + 'A'
      00000004 00000042
      2
      3 00000008 E1A00000             * Back in original file
      7                                     NOP
      8 00000014 44                   * Back in original file
      00000015 6F                     .string "Done"
      00000016 6E
      00000017 65
```

.loop/.break/.endloop*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          1          .eval      0,x
2          2          COEF  .loop
3          3          .word      x*100
4          4          .eval      x+1, x
5          5          .break      x = 6
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** [*parameter*₁] [, ... *parameter*_{*n*}]
 model statements or macro directives
 .endm

Description

The **.macro** directive is 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.

macname names the macro. You must place the name in the source statement's label field.

.macro identifies the source statement as the first line of a macro definition. You must place **.macro** in the opcode field.

[*parameters*] are optional substitution symbols that appear as operands for the **.macro** directive.

model statements are instructions or assembler directives that are executed each time the macro is called.

macro directives are used to control macro expansion.

.endm terminates 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:\470 tools\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 `A_DIR` environment variable

For more information about the `-i` option and the `A_DIR`, see section 3.4, *Naming Alternate Directories for Assembler Input*, on page 3-7.

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, `inc4` and `dec4`. The file `inc4.asm` contains the definition of `inc4`, and `dec4.asm` contains the definition of `dec4`.

inc4.asm	dec4.asm
<pre>* Macro for incrementing inc4 .macro reg1, reg2, reg3, reg4 Add reg1, reg1, #1 ADD reg2, reg2, #1 ADD reg3, reg3, #1 ADD reg4, reg4, #1 .endm</pre>	<pre>* Macro for decrementing dec4 .macro reg1, reg2, reg3, reg4 SUB reg1, reg1, #1 SUB reg2, reg2, #1 SUB reg3, reg3, #1 SUB reg4, reg4, #1 .endm</pre>

Use the archiver to create a macro library:

```
ar470 -a mac inc4.asm dec4.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                                     ; Macro call
3 00000000                          inc4  R7, R6, R5, R4
1 00000000 E2877001                  ADD   R7, R7, #1
1 00000004 E2866001                  ADD   R6, R6, #1
1 00000008 E2855001                  ADD   R5, R5, #1
1 0000000c E2844001                  ADD   R4, R4, #1
4
5                                     ; Macro call
6 00000010                          dec4  R0, R1, R2, R3
1 00000010 E2400001                  SUB   R0, R0, #1
1 00000014 E2411001                  SUB   R1, R1, #1
1 00000018 E2422001                  SUB   R2, R2, #1
1 0000001c E2433001                  SUB   R3, R3, #1
```

.mlist/.mnolist

Start/Stop Macro Expansion Listing

Syntax

.mlist
.mnolist

Description

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

- ☐ The **.mlist** directive shows 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 6, *Macro Language*. For more information about `.loop` and `.endloop`, see page 4-53.

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"    ; Invoke STR_3 macro.
1 00000000 3A        .string ":p1:", ":p2:", ":p3:"
    00000001 70
    00000002 31
    00000003 3A
    00000004 3A
    00000005 70
    00000006 32
    00000007 3A
    00000008 3A
    00000009 70
    0000000a 33
    0000000b 3A
6          .mnolist                ; Suppress expansion.
7 0000000c        STR_3 "as", "I", "am"    ; Invoke STR_3 macro.
8          .mlist                  ; Show macro expansion.
9 00000018        STR_3 "as", "I", "am"    ; Invoke STR_3 macro.
1 00000018 3A        .string ":p1:", ":p2:", ":p3:"
    00000019 70
    0000001a 31
    0000001b 3A
    0000001c 3A
    0000001d 70
    0000001e 32
    0000001f 3A
    00000020 3A
    00000021 70
    00000022 33
    00000023 3A
```

.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**, **.sect**, **.state16**, and **.state32** directives also reset local labels. Local labels that are defined within an include file are not valid outside of the local file.

For more information on the use of local labels, see section 3.8.2, *Local Labels*, on page 3-19.

Example

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

```
1 00000000 E3510000 LABEL1: CMP    r1, #0
2 00000004 2A000001          BCS    $1
3 00000008 E2900001          ADDS   r0, r0, #1
4 0000000c 21A0F00E          MOVCS  pc, lr
5 00000010 E4952004 $1:     LDR    r2, [r5], #4
6                                .newblock    ; Undefine $1 to use again.
7 00000014 E0911002          ADDS   r1, r1, r2
8 00000018 5A000000          BPL    $1
9 0000001c E1F01001          MVNS   r1, r1
10 00000020 E1A0F00E $1:     MOV    pc, lr
```

.option

Select Listing Options

Syntax

.option *option*₁[, *option*₂, . . .]

Description

The **.option** directive selects options for the assembler output listing. *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.
- H** limits the listing of `.half` and `.short` directives to one line.
- M** turns off macro expansions in the listing.
- N** turns off listing (same effect as `.nolist`).
- O** turns on listing (same effect as `.list`).
- R** resets the B, H, M, T, and W options (turns off the limits of B, H, M, T, and W).
- T** limits the listing of `.string` directives to one line.
- W** limits the listing of `.int`, `.long`, and `.word` 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 `-ax` option (see page 3-5).

Options *are not* case sensitive.

Example

This example shows how to limit the listings of the `.byte`, `.word`, `.long`, and `.string` directives to one line each.

```
1 *****
2 ** Limit the listing of .byte, .char, .int, .long, **
3 ** .word, and .string directives to 1 line each. **
4 *****
5         .option B, W, T
6 00000000 BD      .byte   -'C', 0B0h, 5
7 00000003 BC      .char   -'D', 0C0h, 6
8 00000008 0000000A .int    10, 35 + 'a', "abc"
9 0000001c AABBCDD  .long   0AABCCDDh, 536 + 'A'
10 00000024 000015AA .word   5546, 78h
11 0000002c 45      .string "Extended Registers"
12
13 *****
14 **              Reset the listing options.              **
15 *****
16         .option R
17 0000003e BD      .byte   -'C', 0B0h, 5
18         0000003f B0
19         00000040 05
20 00000041 BC      .char   -'D', 0C0h, 6
21         00000042 C0
22         00000043 06
23 00000044 0000000A .int    10, 35 + 'a', "abc"
24         00000048 00000084
25         0000004c 00000061
26         00000050 00000062
27         00000054 00000063
28 00000058 AABBCDD  .long   0AABCCDDh, 536 + 'A'
29         0000005c 00000259
30 00000060 000015AA .word   5546, 78h
31         00000064 00000078
32 00000068 45      .string "Extended Registers"
33         00000069 78
34         0000006a 74
35         0000006b 65
36         0000006c 6E
37         0000006d 64
38         0000006e 65
39         0000006f 64
40         00000070 20
41         00000071 52
42         00000072 65
43         00000073 67
44         00000074 69
45         00000075 73
46         00000076 74
47         00000077 65
48         00000078 72
49         00000079 73
```

.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:

```
TMS470 COFF Assembler      Version x.xx      Mon Jan 27 09:34:58 1997
Copyright (c) 1996-1997    Texas Instruments Incorporated

**** Page Directive Example ****                                PAGE      1
      2                  ;      .
      3                  ;      .
      4                  ;      .

TMS470 COFF Assembler      Version x.xx      Mon Jan 27 09:34:58 1997
Copyright (c) 1996-1997    Texas Instruments Incorporated

**** Page Directive Example ****                                PAGE      2
```

.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 name is significant to eight 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 two special-purpose sections, Sym_Defs and Vars, and assembles code into them.

```
1          *****
2          **      Begin assembling into .text section.      **
3          *****
4 00000000          .text
5 00000000 E3A00078      MOV      R0, #78h
6 00000004 E2801078      ADD      R1, R0, #78h
7          *****
8          **      Begin assembling into Sym_Defs section.    **
9          *****
10 00000000          .sect      "Sym_Defs"
11 00000000 3D4CCCCD      .float  0.05          ; Assembled into Sym_Defs
12 00000004 000000AA X:      .word  0AAh          ; Assembled into Sym_Defs
13 00000008 E2833028      ADD      R3, R3, #28h    ; Assembled into Sym_Defs
14          *****
15          **      Begin assembling into Vars section.        **
16          *****
17 00000000          .sect      "Vars"
18          00000010 WORD_LEN      .set      16
19          00000020 DWORD_LEN     .set      WORD_LEN * 2
20          00000008 BYTE_LEN      .set      WORD_LEN / 2
21          *****
22          **      Resume assembling into .text section.      **
23          *****
24 00000008          .text
25 00000008 E2802042      ADD      R2, R0, #42h    ; Assembled into .text
26 0000000c 03          .byte     3, 4          ; Assembled into .text
27          0000000d 04
28          *****
29          **      Resume assembling into Vars section.        **
30          *****
31 00000000          .sect      "Vars"
32 00000000 000D0000      .field   13, WORD_LEN
33 00000000 000D0A00      .field   0Ah, BYTE_LEN
34 00000004 00000008      .field   10q, DWORD_LEN
```

.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 page 4-44). 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 AR1 and use **
3 **              it instead of the register.          **
4 *****
5 00000001 AUX_R1 .set    R1
6 00000000 E3A01056      MOV    AUX_R1, #56h
7
8 *****
9 **      Set symbol index to an integer expression.  **
10 **              and use it as an immediate operand.  **
11 *****
12 00000035 INDEX .equ    100/2 +3
13 00000004 E2810035      ADD    R0, AUX_R1, #INDEX
14
15 *****
16 **      Set symbol SYMTAB to a relocatable expression. **
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 INDEX  **
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

.space *size in bytes*

.bes *size in bytes*

Description

The **.space** and **.bes** directives reserve the bytes given by the *size in bytes* in the current section and fill them with 0s.

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          *****
7          **   Reserve 0F0 bytes in the .text section.   **
8          *****
9 00000000          .space 0F0h
10 000000f0 00000100      .word 100h, 200h
11 000000f4 00000200
12          *****
13          **   Begin assembling into the .data section.   **
14          *****
14 00000000          .data
15 00000000 49          .string "In .data"
16 00000001 6E
17 00000002 20
18 00000003 2E
19 00000004 64
20 00000005 61
21 00000006 74
22 00000007 61
23          *****
24          ** Reserve 100 bytes in the .data section; RES_1 **
25          **   points to the first byte that contains     **
26          **   reserved bytes.                             **
27          *****
28 21 00000008      RES_1: .space 100
29 22 0000006c 0000000F      .word 15
30 23 00000070 00000008"      .word RES_1
31          *****
32          ** Reserve 20 bytes in the .data section; RES_2 **
33          **   points to the last byte that contains     **
34          **   reserved bytes.                             **
35          *****
36 30 00000087      RES_2: .bes 20
37 31 00000088 00000036      .word 36h
38 32 0000008c 00000087"      .word RES_2
```

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 specified.

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                                ADDL    .macro  dest, src
2                                .global  reset_ctr
3                                ADDS     dest, dest, src
4                                BLCS     reset_ctr
5                                .endm
6
7 00000000                        ADDL    R4, R5
1                                .global  reset_ctr
1 00000000 E0944005                ADDS     R4, R4, R5
1 00000004 2BFFFFFFD!              BLCS     reset_ctr
8 00000008 E5954000                LDR      R4, [R5]
9 0000000c                        ADDL    R0, R4
1                                .global  reset_ctr
1 0000000c E0900004                ADDS     R0, R0, R4
1 00000010 2BFFFFFFFA!             BLCS     reset_ctr
10
11                                .sslist
12
13 00000014 E5B53004                LDR      R3, [R5, #4]!
14 00000018 E5954000                LDR      R4, [R5]
15 0000001c                        ADDL    R4, R3
1                                .global  reset_ctr
1 0000001c E0944003                ADDS     dest, dest, src
#                                ADDS     R4, R4, R3
1 00000020 2BFFFFFFF6!             BLCS     reset_ctr
```

.state16*Assemble 16-Bit Instructions*

Syntax**.state16****Description**

By default, the assembler begins assembling all instructions in a file as 32-bit instructions. Use the **.state16** directive to direct the assembler to begin assembling all instructions at that point as 16-bit instructions. This directive and the **.state32** directive (see page 4-68) allow you to switch between the two assembly modes. If you want to assemble an entire file as 16-bit instructions, use the `-mt` assembler option, which instructs the assembler to begin the assembly process, assembling all instructions as 16-bit instructions.

The **.state16** directive performs an implicit halfword alignment before any instructions are written to the section to ensure that all 16-bit instructions are halfword-aligned. The **.state16** directive also resets any local labels defined.

Example

In this example, the assembler assembles 16-bit instructions, begins assembling 32-bit instructions, and returns to assembling 16-bit instructions.

```
1          .global glob1, glob2
2          ****
3          **      Begin assembling 16-bit instructions.      **
4          ****
5 00000000          .state16
6
7 00000000 4808      LDR      r0, glob1_a
8 00000002 4909      LDR      r1, glob2_a
9 00000004 6800      LDR      r0, [r0]
10 00000006 6809      LDR      r1, [r1]
11 00000008 0080      LSL      r0, r0, #2
12 0000000a 3156      ADD      r1, #56h
13 0000000c 4778      BX      pc
14 0000000e 46C0      NOP
15          ****
16          **      Switch to 32-bit instructions to use the    **
17          **      32-bit state long multiply instruction.      **
18          ****
19 00000010          .state32
20
21 00000010 E0845190  UMULL    r5, r4, r0, r1
22 00000014 E28FE001  ADD      lr, pc, #1
23 00000018 E12FFF1E  BX      lr
24          ****
25          **      Continue assembling 16-bit instructions.    **
26          ****
27 0000001c          .state16
28
29 0000001c 1A2D      SUB      r5, r5, r0
30 0000001e D200      BCS      $1
31 00000020 3C01      SUB      r4, #1
32 00000022          $1
33 00000024 00000000! glob1_a .word  glob1
34 00000028 00000000! glob2_a .word  glob2
```

.state32*Assemble 32-Bit Instructions***Syntax****.state32****Description**

By default, the assembler begins assembling all instructions in a file as 32-bit instructions. When you use the `-mt` assembler option or the `.state16` directive to assemble 16-bit instructions, you can use the **.state32** directive to tell the assembler to begin assembling all instructions after the `.state32` directive as 32-bit instructions.

The `.state32` directive performs an implicit word alignment before any instructions are written to the section to ensure that all 32-bit instructions are word-aligned. The `.state32` directive also resets any local labels defined.

Example

In this example, the assembler assembles 32-bit instructions, begins assembling 16-bit instructions, and returns to assembling 32-bit instructions.

```

1          .global globs, filter
2          ****
3          **      Begin assembling 32-bit instructions.      **
4          ****
5          .state32
6          00000000 E28F4001      ADD     r4, pc, #1
7          00000004 E12FFF14      BX      r4
8          ****
9          **      Switch to 16-bit instructions to use      **
10         **      less code space.                          **
11         ****
12         .state16
13         00000008 2200          MOV     r2, #0
14         0000000a 2300          MOV     r3, #0
15         0000000c 4C0B          LDR     r4, globs_a
16         0000000e 2500          MOV     r5, #0
17         00000010 2600          MOV     r6, #0
18         00000012 2700          MOV     r7, #0
19         00000014 4690          MOV     r8, r2
20         00000016 4691          MOV     r9, r2
21         00000018 4692          MOV     r10, r2
22         0000001a 4693          MOV     r11, r2
23         0000001c 4694          MOV     r12, r2
24         0000001e 4695          MOV     r13, r2
25         00000020 4778          BX      pc
26         00000022 46C0          NOP
27         ****
28         **      Continue assembling 32-bit instructions.    **
29         ****
30         .state32
31         00000024 E4940004      LDR     r0, [r4], #4
32         00000028 E5941000      LDR     r1, [r4]
33         0000002c EBFFF3F3!     BL      filter
34         00000030 E1500001      CMP     r0, r1
35         00000034 30804005      ADDCC   r4, r0, r5
36         00000038 20464001      SUBCS   r4, r6, r1
37         0000003c 00000000! globs_a .word globs

```

.string*Initialize Text*

Syntax**.string** {*expr*₁ | "*string*₁" } [, ... , {*expr*_{*n*} | "*string*_{*n*}" }]**Description**

The **.string** directive places 8-bit characters from a character string into consecutive bytes in the current section. Each *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 assembler truncates any values that are greater than eight bits. You may have up to 100 operands, but they must fit on a single source statement line.

If you use a label, 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 about **.struct/.endstruct**, see page 4-70.

Example

This example shows 8-bit values placed into words in the current section.

```
1 00000000 41      Str_Ptr:  .string  "ABCD"
   00000001 42
   00000002 43
   00000003 44
2 00000004 41      .string  41h, 42h, 43h, 44h
   00000005 42
   00000006 43
   00000007 44
3 00000008 41      .string  "Austin", "Houston", "Dallas"
   00000009 75
   0000000a 73
   0000000b 74
   0000000c 69
   0000000d 6E
   0000000e 48
   0000000f 6F
   00000010 75
   00000011 73
   00000012 74
   00000013 6F
   00000014 6E
   00000015 44
   00000016 61
   00000017 6C
   00000018 6C
   00000019 61
   0000001a 73
4 0000001b 30      .string  36 + 12
```

**.struct/
.endstruct/.tag**

Declare Structure Type

Syntax

[stag]	.struct	[expr]
[mem ₀]	<i>element</i>	[expr ₀]
[mem ₁]	<i>element</i>	[expr ₁]
.	.	.
.	.	.
.	.	.
[mem _n]	.tag stag	[expr _n]
.	.	.
.	.	.
.	.	.
[mem _N]	<i>element</i>	[expr _N]
[size]	.endstruct	
<i>label</i>	.tag	stag

Description

The **.struct** 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 **.struct** directive does not allocate memory; it merely creates a symbolic template that can be used repeatedly.

The **.endstruct** directive terminates the structure definition.

The **.tag** directive gives structure characteristics to a *label*, simplifying the symbolic representation and providing the ability to define structures that contain other structures. The **.tag** directive does not allocate memory. The structure tag (stag) of a **.tag** directive must have been previously defined.

Following are descriptions of the parameters used with the **.struct**, **.endstruct**, and **.tag** directives:

- ☐ The *element* is one of the following descriptors: **.string**, **.byte**, **.char**, **.int**, **.half**, **.short**, **.word**, **.long**, **.double**, **.float**, **.tag**, or **.field**. All of these except **.tag** are typical directives that initialize memory. Following a **.struct** directive, these directives describe the structure element's size. They do not allocate memory. A **.tag** directive is a special case because stag must be used (as in the definition of stag).
- ☐ The *expr* is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.
- ☐ The *expr_{N/n}* 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 *mem_{N/n}* 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.
- ❑ The *size* is an optional label for the total size of the structure.
- ❑ The *stag* is the structure's tag. Its value is associated with the beginning of the structure. If no stag 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. An stag is optional for .struct, but is required for .tag.

Note: Directives That Can Appear in a .struct/.endstruct Sequence

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.

These 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 = 4
5
6 00000000 E59F0004      LDR R0, REAL_A
7 00000004 E5904004      LDR R4, [R0, #REAL_REC.DEN]
8 00000008 E0811004      ADD R1, R1, R4
9 00000000              .bss REAL, REAL_LEN ; allocate mem rec
10 0000000c 00000000- REAL_A .word REAL
11
```

Example 2

```
12          CPLX_REC  .struct
13          00000000  REALI   .tag REAL_REC ; stag
14          00000008  IMAGI   .tag REAL_REC ; member1 = 0
15          00000010  CPLX_LEN .endstruct   ; cplx_len = 8
16
17          COMPLEX    .tag CPLX_REC        ; assign structure
18                                     ; attribute
19 00000010          COMPLEX .space CPLX_LEN ; allocate space
20 00000020 E51F4018      LDR R4, COMPLEX.REALI ; access structure
21 00000024 E0811004      ADD R1, R1, R4
```

File Name	File Type	File Size	File Date	File Location
1. tab	Table	1.0 MB	2023-10-27	Table

Example 3

```

1          .struct          ; no stag puts mems into
2                               ; global symbol table
3      00000000 X      .int      ; create 3 dim templates
4      00000004 Y      .int
5      00000008 Z      .int
6      0000000C      .endstruct

```

Example 4

```

1          BIT_REC      .struct                      ; stag
2          00000000      STREAM      .string 64
3          00000040      BIT7        .field 7          ; bit7 = 64
4          00000040      BIT8        .field 9          ; bit9 = 64
5          00000042      BIT10       .field 10         ; bit10 = 64
6          00000044      X_INT       .int              ; x_int = 68
7          00000048      BIT_LEN     .endstruct        ; length = 72
8
9 00000000      BITS      .space BIT_LEN
10             BITS      .tag BIT_REC
11 00000048 E51F0010      LDR R0, BITS,BIT7
12 0000004c E0811000      ADD R1, R1, R0

```

.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 E1A00000                NOP
3 00000004 E1A00000                NOP
4 00000008 E1A00000                NOP
5
6
7 0000000c E1A00000                NOP
8 00000010 E1A00000                NOP
9 00000014 E1A00000                NOP
10
11
12 00000018 E1A00000                NOP
13 0000001c E1A00000                NOP
14 00000020 E1A00000                NOP
```

.text

Assemble Into .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.

.text is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you use a .data to .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 0A          .byte  0Ah, 0Bh
6 00000001 0B
7
8          *****
9          ** Begin assembling into .text section. **
10         *****
11         .text
12 START:  .string "A","B","C"
13         00000001 42
14         00000002 43
15         00000003 58      END:  .string "X","Y","Z"
16         00000004 59
17         00000005 5A
18 00000008 E3A01003      MOV    R1, #END-START
19 0000000c E1A01181      MOV    R1, R1, LSL #3
20
21         *****
22         ** Resume assembling into .data section.**
23         *****
24 00000002          .data
25 00000002 0C          .byte  0Ch, 0Dh
26 00000003 0D
27
28         *****
29         ** Resume assembling into .text section.**
30         *****
31         .text
32 00000010          .string "QUIT"
33 00000011 55
34 00000012 49
35 00000013 54
```

.title*Define Page Title*

Syntax

.title "string"

Description

The **.title** 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.

The *string* 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:

```
***WARNING! line x:w0001: String is too long-will be truncated
```

The assembler prints the title on the page that follows the directive and on subsequent pages until another **.title** directive is processed. If you want a title on the first page, the first source statement must contain a **.title** directive.

Example

In this example, one title is printed on the first page and a different title is printed on succeeding pages.

Source file:

```
                .title  "**** Fast Fourier Transforms ****"
;               .
;               .
;               .
                .title  "**** Floating-Point Routines ****"
                .page
```

Listing file:

```
TMS470 COFF Assembler      Version                      Mon Jan 27 09:47:10 1997
Copyright (c) 1996-1997    Texas Instruments Incorporated

**** Fast Fourier Transforms ****                                PAGE      1

      2                      ;          .
      3                      ;          .
      4                      ;          .
TMS470 COFF Assembler      Version                      Mon Jan 27 09:47:10 1997
Copyright (c) 1996-1997    Texas Instruments Incorporated

**** Floating-Point Routines ****                                PAGE      2
```

.usect*Reserve Uninitialized Space*

Syntax

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

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, independent 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 word *alignment* is an optional parameter. It ensures that the space allocated to the symbol occurs on the specified boundary. The boundary indicates the size of the alignment in bytes and can be set to a power of 2 between 2^0 and 2^{15} , inclusive. If the SPC is aligned at the specified boundary, it is not incremented.

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*.

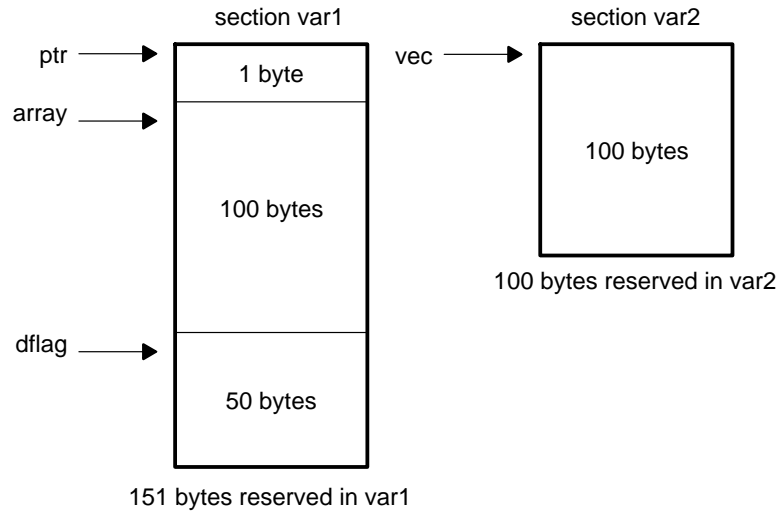
Example

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 in a block of 50 bytes in `var1`. The symbol `vec` 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 the .text section.          **
3          ****
4 00000000          .text
5 00000000 E3A01003      MOV      R1, #03h
6
7          ****
8          **          Reserve 1 byte in the var1 section.          **
9          ****
10 00000000      ptr      .usect  "var1", 1
11
12          ****
13          **          Reserve 100 bytes in the var1 section.          **
14          ****
15 00000001      array     .usect  "var1", 100
16
17 00000004 E281001F      ADD      R0, R1, #037 ; Still in .text
18
19          ****
20          **          Reserve 50 bytes in the var1 section.          **
21          ****
22 00000065      dflag     .usect  "var1", 50
23
24 00000008 E2812064      ADD      R2, R1, #dflag - array ; Still in .text
25
26          ****
27          **          Reserve 100 bytes in the var2 section.          **
28          ****
29 00000000      vec       .usect  "var2", 100
30
31 0000000c E0824000      ADD      R4, R2, R0 ; Still in .text
32          ****
33          **          Declare a .usect symbol to be external.          **
34          ****
35          .global array
```

Figure 4–8. The .usect Directive



.var

Use Substitution Symbols as Local Variables

Syntax

.var *sym*₁ [*sym*₂, ... , *sym*_{*n*}]

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 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

This chapter describes macro directives, substitution symbols used as macro parameters, and how to create macros.

Topic	Page
5.1 Using Macros	5-2
5.2 Defining Macros	5-3
5.3 Macro Parameters/Substitution Symbols	5-5
5.4 Macro Libraries	5-13
5.5 Using Conditional Assembly in Macros	5-14
5.6 Using Labels in Macros	5-16
5.7 Producing Messages in Macros	5-17
5.8 Using Directives to Format the Output Listing	5-19
5.9 Using Recursive and Nested Macros	5-21
5.10 Macro Directives Summary	5-23

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 Symbol*, on page 5-5, for more information.

Using a macro is a three-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:

- ☐ Macros can be defined at the beginning of a *source file* or in a copy/include file. See section 5.2, *Defining Macros*, for more information.
- ☐ 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) contains 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*, on page 5-13.

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 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 `.mno` directive. For more information, see section 5.8, *Formatting the Output Listing*, on page 5-19.

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 page 4-32); they can also be defined in a macro library. For more information, see section 5.4, *Macro Libraries*, on page 5-13.

Macro definitions can be nested, and they can call other macros, but all elements of any macro must be defined in the same file. Nested macros are discussed in section 5.9, *Using Recursive and Nested Macros*, on page 5-21.

A macro definition is a series of source statements in the following format:

```

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

```

<i>macname</i>	names the macro. You must place the name in the source statement's label field. Only the first 32 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.
.macro	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</i> ₁ , <i>parameter</i> _{<i>n</i>}	are optional substitution symbols that appear as operands for the .macro directive. Parameters are discussed in section 5.3, <i>Macro Parameters/Substitution Symbols</i> , on page 5-5.
<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.
.mexit	is a directive that functions as a <i>goto .endm</i> . The .mexit directive is useful when error testing confirms that macro expansion will fail and completing the rest of the macro is unnecessary.
.endm	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, `add3`, with four parameters:

```
1          *
2
3          *      add3
4          *
5          *      ADDR = P1 + P2 + P3
6
7          add3   .macro P1, P2, P3, ADDR
8
9                  ADD    ADDR, P1, P2
10                 ADD    ADDR, ADDR, P3
11                 .endm
```

Macro call: The following code calls the `add3` macro with four arguments:

```
12
13 00000000          add3 R1, R2, R3, R0
```

Macro expansion: The following code shows the substitution of the macro definition for the macro call. The assembler substitutes `R1`, `R2`, `R3`, and `R0` for `P1`, `P2`, `P3`, and `ADDR`.

```
1
1      00000000 E0810002      ADD    R0, R1, R2
1      00000004 E0800003      ADD    R0, R0, R3
```

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*, on page 5-17, 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*, on page 3-25).

Valid substitution symbols can be up to 32 characters long and *must begin with a letter*. The remainder of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

Substitution symbols used as macro parameters are local to the macro they are defined in. You can define up to 32 local substitution symbols per macro, including substitution symbols defined with the `.var` directive. For more information about the `.var` directive, see section 5.3.6, *Substitution Symbols as Local Variables in Macros*, on page 5-12.

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 enclose the arguments in 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:

```

Parms 100,label
;      a = 100
;      b = label
;      c = " "

```

```

Parms 100,label,x,y
;      a = 100
;      b = label
;      c = x,y

```

```

Parms 100, , x
;      a = 100
;      b = " "
;      c = x

```

```

Parms "100,200,300",x,y
;      a = 100,200,300
;      b = x
;      c = y

```

```

Parms ""string"" ,x,y
;      a = "string"
;      b = x
;      c = y

```

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.

The syntax of the **.asg** directive is:

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

Example 5–3 shows character strings being assigned to substitution symbols.

Example 5–3. The .asg Directive

```

.asg R13, stack_ptr      ; stack pointer
.asg R14, ret_reg        ; link register
.asg R15, prog_ctr       ; program counter
.asg ^, XOR              ; bitwise exclusive or

```

- The **.eval** directive performs arithmetic on numeric substitution symbols.

The syntax of the **.eval** directive is

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

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.

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, the **.eval** directive evaluates an expression and then assigns the character string equivalent to a substitution symbol.

5.3.2 Built-In Substitution Symbol Functions

The following built-in substitution symbol functions enable you to make decisions based on 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
\$\$symlen(a)	Length of string <i>a</i>
\$\$symcmp(a,b)	< 0 if <i>a</i> < <i>b</i> ; 0 if <i>a</i> = <i>b</i> ; > 0 if <i>a</i> > <i>b</i>
\$\$firstch(a,ch)	Index of the first occurrence of character constant <i>ch</i> in string <i>a</i>
\$\$lastch(a,ch)	Index of the last occurrence of character constant <i>ch</i> in string <i>a</i>
\$\$isdefed(a)	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
\$\$ismember(a,b)	Top member of list <i>b</i> is assigned to string <i>a</i> 0 if <i>b</i> is a null string
\$\$iscons(a)	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
\$\$isname(a)	1 if string <i>a</i> is a valid symbol name 0 if string <i>a</i> is not a valid symbol name
\$\$isreg(a)[†]	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

[†] For more information about predefined register names, see section 3.8.5, *Predefined Symbolic Constants*, on page 3-23.

Example 5–5 shows built-in substitution symbol functions.

Example 5–5. Using Built-In Substitution Symbol Functions

```
.asg    label, ADDR                ; ADDR = label
.if     ($$symcmp(ADDR, "label") = 0) ; evaluates to true
LDR     R4, ADDR
.endif
.asg    "x,y,z" , list             ; list = x,y,z
.if     ($$ismember(ADDR,list))    ; ADDR = x, list = y,z
SUB     R4, R4, #4                 ; sub x
.endif
```


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"
LDR   R0, x
* LDR   R0, x ; 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 in 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 enclosed in colons before it expands 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

```
1          force .macro
2          .asg  0,x
3          .loop  8
4          AUX:x: .set  x
5          .eval  x+1,x
6          .endloop
7          .endm
8
9 00000000      force
1         .asg  0,x
1         .loop  8
1         AUX:x: .set  x
1         .eval  x+1,x
1         .endloop
2         00000000 AUX0 .set  0
2         .eval  0+1,x
2         00000001 AUX1 .set  1
2         .eval  1+1,x
2         00000002 AUX2 .set  2
2         .eval  2+1,x
2         00000003 AUX3 .set  3
2         .eval  3+1,x
2         00000004 AUX4 .set  4
2         .eval  4+1,x
2         00000005 AUX5 .set  5
2         .eval  5+1,x
2         00000006 AUX6 .set  6
2         .eval  6+1,x
2         00000007 AUX7 .set  7
2         .eval  7+1,x
```

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, expression₁ represents the substring's starting position, and expression₂ 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

```

ADDX      .macro      dst, imm
          .var         TMP
          .asg         :imm(1):, TMP
          .if          $$symcmp(TMP, "#") = 0
ADD        dst, dst, imm
          .else
          .emsg         "Bad Macro Parameter"
          .endif
          .endm

ADDX      R9, #100           ; macro call
ADDX      R9, R8             ; macro call

```

In Example 5–8, subscripted substitution symbols redefine the ADD instruction so that it handles short immediate values.

Example 5–9. Using Subscripted Substitution Symbols to Find Substrings

```
substr .macro      start, strg1, strg2, pos
      .var         LEN1, LEN2, I, TMP
      .if          $$symlen(start) = 0
      .eval        1, start
      .endif
      .eval        0, pos
      .eval        1, i
      .eval        $$symlen(strg1), LEN1
      .eval        $$symlen(strg2), LEN2
      .loop
      .break       i = (LEN2 - LEN1 + 1)
      .asg         ":strg2(I, LEN1):", TMP
      .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 after expansion they are lost.

```
.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 detail on page 4-55). 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*, on page 5-2, 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*, on page 5-19 and the .mlist description on page 4-57. 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 can 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 well-defined expression]  
.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. For more information on the **.if/.elseif/.else/.endif** directives, see page 4-47.

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). For more information on the **.loop/.break/.endloop** directives, see page 4-53.

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             ; force break
.endif

.eval   x+1,x
.endloop

```

Example 5–12. Built-In Substitution Symbol Functions Used in a Conditional Assembly Code Block

```

.fcncolist
*
*Double Add or Subtract
*
DBL    .macro ABC, dsth, dstl, srch, srcl ; add or subtract double

        .if      $$symcmp(ABC,"+")
        ADDS     dstl, dstl, srcl          ; add double
        ADC      dsth, dsth, srch

        .elseif  $$symcmp(ABC,"-")
        SUBS     dstl, dstl, srcl          ; subtract double
        SUBS     dsth, dsth, srch

        .else
        .emsg    "Incorrect Operator Parameter"

        .endif

        .endm

*Macro Call
DBL    -, R4, R5, R6, R7

```

For more information, see section 4.8, *Directives That Enable Conditional Assembly*, on page 4-20.

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 the label with a question mark, and the assembler replaces the question mark with 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          ; define macro to find minimum
2          MIN      .macro dst, src1, src2
3                  CMP      src1, src2
4                  BCC      m1?
5                  MOV      dst, src1
6                  B        m2?
7
8          m1?      MOV      dst, src2
9          m2?
10         .endm
11
12         ; call macro
13 00000000      .state16
14 00000000      MIN      r4, r1, r2
1 00000000 4291      CMP      r1, r2
1 00000002 D301      BCC      m1?
1 00000004 1C0C      MOV      r4, r1
1 00000006 E000      B        m2?
1
1 00000008 1C14      m1?      MOV      r4, r2
1 0000000a      m2?
```

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 page 3-5).

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

```

1          MUL_I    .macro x,y
2                  .if ($$symlen(x) ==0)
3                      .emsg "ERROR -- Missing Parameter"
4                      .mexit
5                  .elseif ($$symlen(y) == 0)
6                      .emsg "ERROR -- Missing Parameter"
7                      .mexit
8                  .else
9                      MOV  R1, x
10                     MOV  R2, y
11                     MUL  R0, R1, R2
12                 .endif
13                 .endm
14
15 00000000      MUL_I #50, #51
1                  .if ($$symlen(x) ==0)
1                      .emsg "ERROR -- Missing Parameter"
1                      .mexit
1                  .elseif ($$symlen(y) == 0)
1                      .emsg "ERROR -- Missing Parameter"
1                      .mexit
1                  .else
1                      MOV  R1, #50
1                      MOV  R2, #51
1                      MUL  R0, R1, R2
1                  .endif
16
17 00000000c      MUL_I
1                  .if ($$symlen(x) ==0)
1                      .emsg "ERROR -- Missing Parameter"
1                      .emsg "ERROR -- Missing Parameter"
1          ***** USER ERROR ***** - : ERROR -- Missing Parameter
1                      .mexit
1
1 Error, No Warnings

```

5.8 Using Directives to Format the Output Listing

Macros, substitution symbols, and conditional assembly directives can 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 output list 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 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 data memory address `loc`. The `fact` macro accomplishes this by calling `fact1`, which calls itself recursively.

Example 5–16. Using Recursive Macros

```
fact      .macro N, loc          ; N is an integer constant.
                                   ; Register loc address = N!
                                   ; 0! = 1! = 1
        .if      N < 2
        MOV      loc, #1

        .else
        MOV      loc, #N          ; N >= 2 so, store N in loc.
        .eval    N-1, N          ; Decrement N, and do the
                                   ; factorial of N - 1.
        fact1
                                   ; Call fact with current
                                   ; environment.
        .endm

fact1     .macro
        .if      N > 1
        MOV      R0, #N          ; N > 1 so, store N in R0.
        MUL      loc, R0, loc    ; Multiply present factorial
                                   ; by present position.
        .eval    N - 1, N        ; Decrement position.
        fact1
                                   ; Recursive call.
        .endif
        .endm
```

5.10 Macro Directives Summary

The following directives can be used with macros. The `.macro`, `.mexit`, `.endm` and `.var` directives are only valid with macros; the remaining directives are general assembly language directives.

Table 5–2. Creating Macros

Mnemonic and Syntax	Description	See Page	
		Macro Use	Directive Description
<code>macname .macro</code> [<i>parameter</i> ₁] [, ... , <i>parameter</i> _n]	Define macro by <i>macname</i>	5-3	5-3
<code>.mlib</code> <i>filename</i>	Identify library containing macro definitions	5-13	4-55
<code>.mexit</code>	Go to <code>.endm</code>	5-3	5-3
<code>.endm</code>	End macro definition	5-3	5-3

Table 5–3. Manipulating Substitution Symbols

Mnemonic and Syntax	Description	See Page	
		Macro Use	Directive Description
<code>.asg</code> [" <i>character string</i> "], <i>substitution symbol</i>	Assign character string to substitution symbol	5-6	4-25
<code>.eval</code> <i>well-defined expression</i> , <i>substitution symbol</i>	Perform arithmetic on numeric substitution symbols	5-7	4-25
<code>.var</code> <i>sym</i> ₁ [, <i>sym</i> ₂ , ... , <i>sym</i> _n]	Define local macro symbols	5-12	5-12

Table 5–4. Conditional Assembly

Mnemonic and Syntax	Description	See Page	
		Macro Use	Directive Description
<code>.if</code> <i>well-defined expression</i>	Begin conditional assembly	5-14	4-47
<code>.elseif</code> <i>well-defined expression</i>	Optional conditional assembly block	5-14	4-47
<code>.else</code>	Optional conditional assembly block	5-14	4-47
<code>.endif</code>	End conditional assembly	5-14	4-47
<code>.loop</code> [<i>well-defined expression</i>]	Begin repeatable block assembly	5-14	4-53

Mnemonic and Syntax	Description	Macro Use	Directive Description
.break [<i>well-defined expression</i>]	Optional repeatable block assembly	5-14	4-53
.endloop	End repeatable block assembly	5-14	4-53

Table 5–5. Producing Assembly-Time Messages

Mnemonic and Syntax	Description	See Page	
		Macro Use	Directive Description
.emsg	Send error message to standard output	5-17	4-37
.mmsg	Send assembly-time message to standard output	5-17	4-37
.wmsg	Send warning message to standard output	5-17	4-37

Table 5–6. Formatting the Listing

Mnemonic and Syntax	Description	See Page	
		Macro Use	Directive Description
.drlist	Print all directive lines to the listing file (default)	5-19	4-36
.drnolist	Suppress the printing of certain directives in listing file	5-19	4-36
.fclist	Allow false conditional code block listing (default)	5-19	4-40
.fcnolist	Suppress false conditional code block listing	5-19	4-40
.mlist	Allow macro listings (default)	5-19	4-57
.mnolist	Suppress macro listings	5-19	4-57
.sslist	Allow expanded substitution symbol listing	5-19	4-66
.ssnolist	Suppress expanded substitution symbol listing (default)	5-19	4-66

Archiver Description

The TMS470R1x 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.

This chapter describes instructions for invoking the archives, creating new archive libraries, and modifying existing libraries.

Topic	Page
6.1 Archiver Overview	6-2
6.2 The Archiver's Role in the Software Development Flow	6-3
6.3 Invoking the Archiver	6-4
6.4 Archiver Examples	6-6

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.

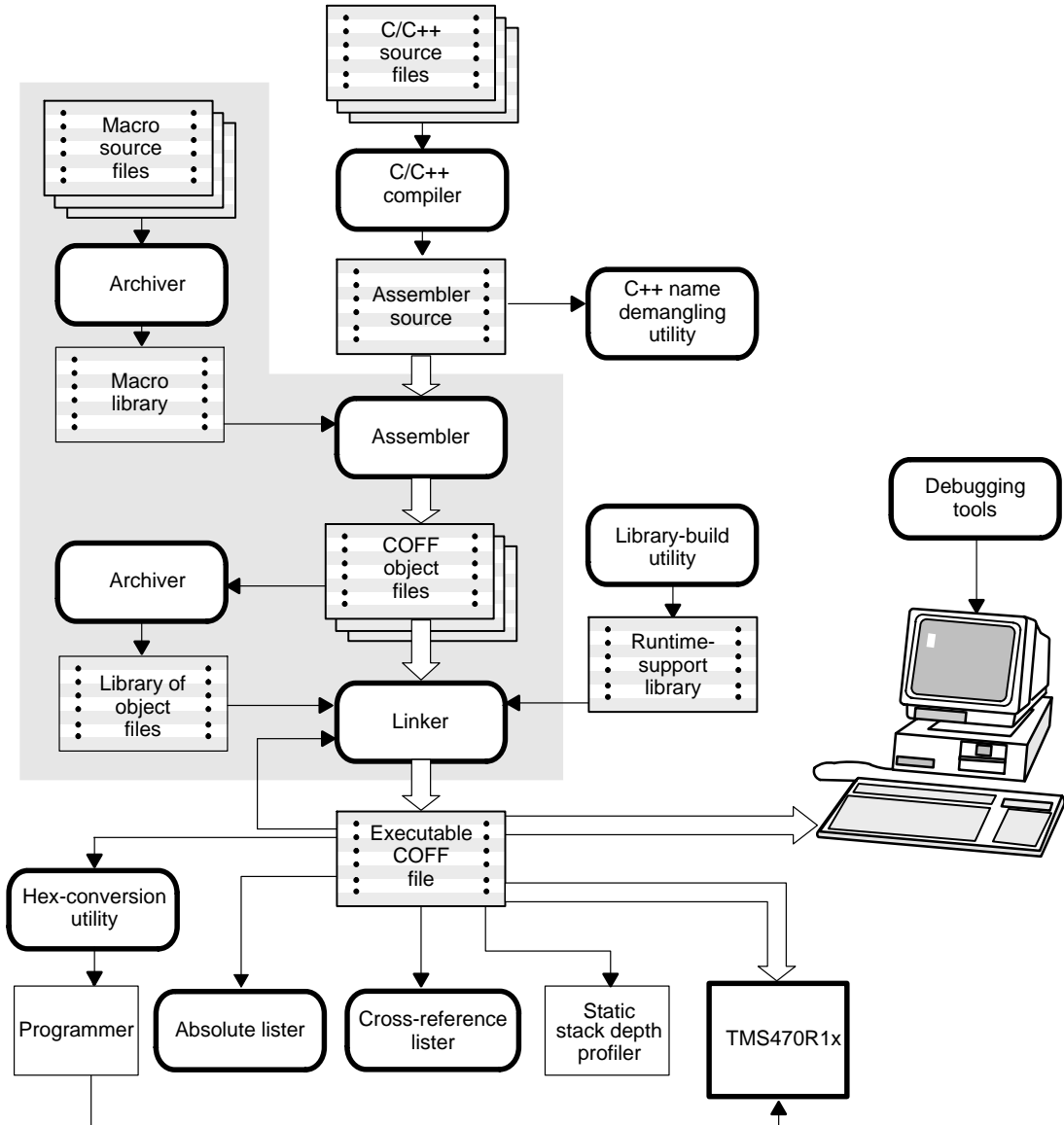
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The 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*, describes 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 illustrates the archiver's role in the software development process. Both the assembler and the linker accept libraries as input.

Figure 6–1. The Archiver in the TMS470R1x Software Development Flow



6.3 Invoking the Archiver

To invoke the archiver, enter:

ar470 *[-]command [options] libname [filename₁ ... filename_n]*

ar470 is the command that invokes the archiver.

[-]command tells the archiver how to manipulate the existing library members and any specified *filenames*. 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 page 6-7 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.
- u** replaces library members only if the replacement has a more recent modification date. You must use the *r* command with the *u* command to specify which members to replace.

- 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).
- 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. If you do not specify an extension for a *filename*, the archiver uses the default extension *.obj*.

Note: Naming Library Members

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 `tan.obj`., specify:

```
ar470 -a function sin.obj cos.obj tan.obj
```

The archiver responds as follows:

```
TMS470 Archiver           Version x.xx
Copyright (c) 1996-1997   Texas Instruments Incorporated
==> new archive 'function.lib'
==> building archive 'function.lib'
```

- ❑ You can print a table of contents of `function.lib` with the `-t` command:

```
ar470 -t function
```

The archiver responds as follows:

```
TMS470 Archiver           Version x.xx
Copyright (c) 1996-1997   Texas Instruments Incorporated
```

FILE NAME	SIZE	DATE
sin.obj	1118	Tue Jan 14 15:00:44 1997
cos.obj	1096	Tue Jan 13 09:00:57 1997
tan.obj	1172	Thu Jan 16 09:45:24 1997

- ❑ If you want to add new members to the library, specify:

```
ar470 -as function atan.obj
```

The archiver responds as follows:

```
TMS470 Archiver           Version x.xx
Copyright (c) 1996-1997   Texas Instruments Incorporated
==> 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`.

```
ar470 -x macros push.asm
```

The archiver makes a copy of `push.asm` and places it in the current directory, but 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:

```
ar470 -r macros push.asm
```

- ❑ If you want to use a command file, specify the command filename after the `@` command. For example:

```
ar470 @modules.cmd
```

The archiver responds as follows:

```
TMS470 Archiver           Version x.xx
Copyright (c) 1996-1997   Texas Instruments Incorporated
==> building archive 'modules.lib'
```

This is the `modules.cmd` command file:

```
; Command file to replace members of the modules
; library with updated files
; First, specify the r command and the u option:
ru
; Next, specify the library name:
modules.lib
; Last, 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
```

In this example, 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.

Linker Description

The TMS470R1x 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
7.1 Linker Overview	7-2
7.2 The Linker's Role in the Software Development Flow	7-3
7.3 Invoking the Linker	7-4
7.4 Linker Options	7-5
7.5 Linker Command Files	7-21
7.6 Object Libraries	7-24
7.7 The MEMORY Directive	7-26
7.8 The SECTIONS Directive	7-30
7.9 Specifying a Section's Run-Time Address	7-44
7.10 Using UNION and GROUP Statements	7-48
7.11 Special Section Types (DSECT, COPY, and NOLOAD)	7-51
7.12 Default Allocation Algorithm	7-52
7.13 Assigning Symbols at Link Time	7-54
7.14 Creating and Filling Holes	7-62
7.15 Linker-Generated Copy Tables	7-66
7.16 Partial (Incremental) Linking	7-81
7.17 Linking C/C++ Code	7-83
7.18 Linker Example	7-87

7.1 Linker Overview

The TMS470R1x 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

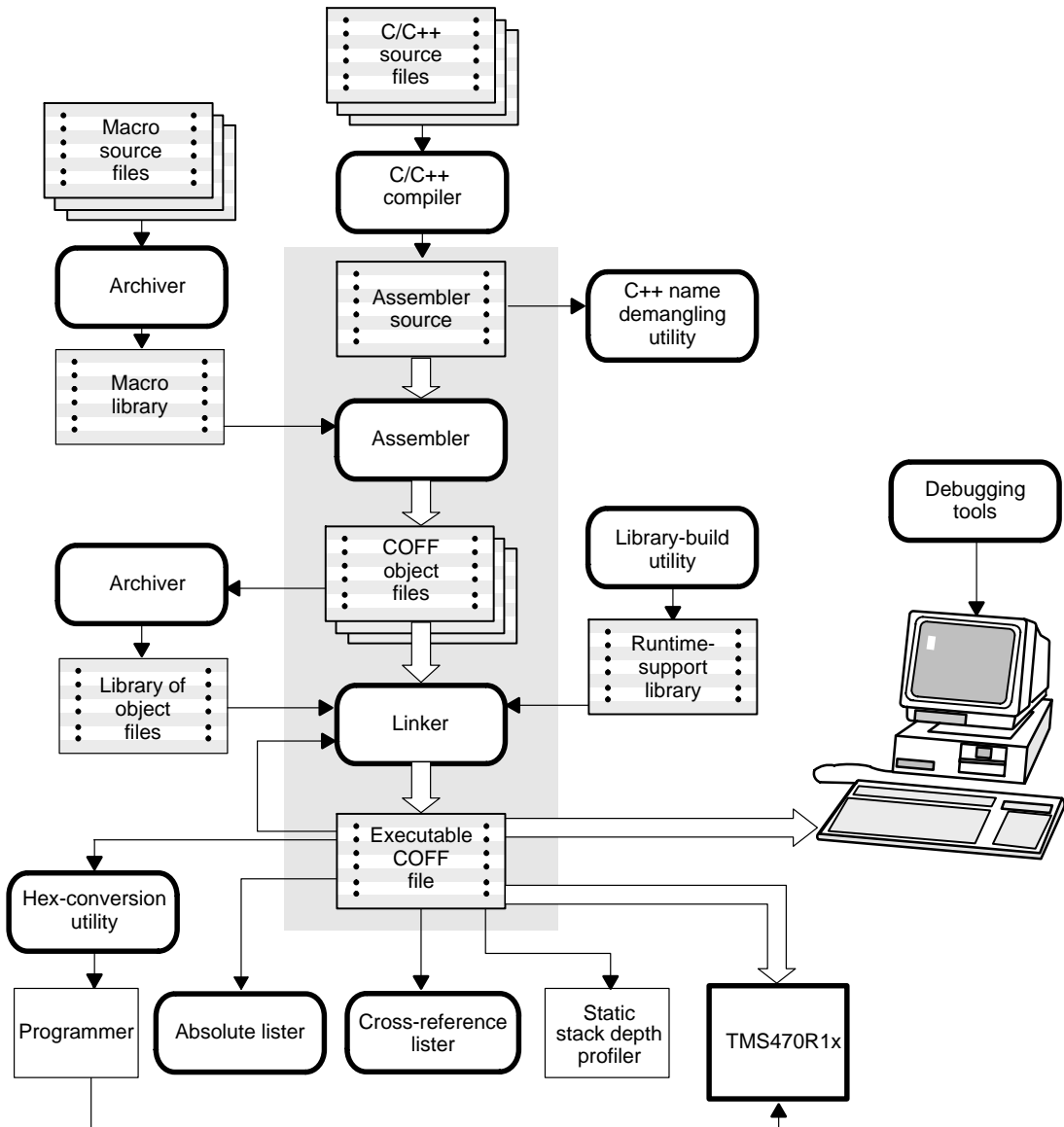
The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow 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 TMS470R1x device.

Figure 7–1. The Linker in the TMS470R1x Software Development Flow



7.3 Invoking the Linker

The general syntax for invoking the linker is:

```
cl470 -z [options] filename1 ... filenamen
```

cl470 -z is the command that invokes the linker.

options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 7.4, *Linker Options*, on page 7-5.)

filename₁,
filename_n can be object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; 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 *a.out*, unless you use the *-o* 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*.

```
cl470 -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, specifying the command filename as an input file:

```
cl470 -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:

```
cl470 -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*, on page 7-83.

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	Page
-a	Produces an absolute, executable module. This is the default; if neither -a nor -r is specified, the linker acts as if -a were specified.	7-6
-abs	Produces an absolute listing file	7-7
-ar	Produces a relocatable, executable object module.	7-6
-args	Allocates memory to be used by the loader to pass arguments	7-8
-b	Disables merge of symbolic debugging information.	7-8
-c	Autoinitializes variables at run time.	7-9
-cr	Autoinitializes variables at load time.	7-9
-e <i>global_symbol</i>	Defines a global <i>symbol</i> that specifies the primary entry point for the output module.	7-9
-f <i>fill_value</i>	Sets the default fill value for holes within output sections; <i>fill_value</i> is a 32-bit constant.	7-10
-g <i>symbol</i>	Makes <i>symbol</i> global (overrides -h).	7-10
-h	Makes all global symbols static.	7-10
-heap <i>size</i>	Sets heap size (for the dynamic memory allocation in C) to <i>size</i> bytes and defines a global symbol that specifies the heap size. Default = 2K bytes.	7-11
-help	Prints out a full listing of switches and their meanings.	--
-i <i>pathname</i> [†]	Alters the library-search algorithm to look in a directory named with <i>pathname</i> before looking in the default location. This option must appear before the -l option.	7-12
-j	Disables conditional linking.	7-13
-l <i>filename</i> [†]	Names an archive library <i>filename</i> as linker input.	7-11
-m <i>filename</i> [†]	Produces a map or listing of the input and output sections, including holes, and places the listing in <i>filename</i> .	7-13
-o <i>filename</i> [†]	Names the executable, output module. The default filename is a.out.	7-15

[†] The *pathname* or *filename* must follow operating system conventions.

Table 7–1. Linker Options Summary (Continued)

Option	Description	Page
-priority	Satisfies unresolved references by the first library that contains a definition for that symbol	7-19
-r	Produces a nonexecutable, relocatable output module.	7-6
-s	Strips symbol table information and line number entries from the output module.	7-15
-stack size	Sets C system stack size to <i>size</i> bytes and defines a global symbol that specifies the stack size. Default = 2K bytes.	7-16
-trampolines	Generates far call trampolines	7-16
-u symbol	Places an unresolved external <i>symbol</i> into the output module's symbol table.	7-18
-w	Displays a message when an undefined output section is created.	7-19
-x	Forces rereading of libraries, which resolves back references.	7-19

† The *pathname* or *filename* must follow operating system conventions.

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.

□ 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 (section 7.13.4, on page 7-57, describes these symbols)
- 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:

```
cl1470 -z -a file1.obj file2.obj
```

Note: –a and –r Options

If you do not use the –a or the –r option, the linker acts as if you specified –a.

❑ 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 will be 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`:

```
cl470 -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*, on page 7-81.

❑ 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`:

```
cl470 -z -ar file1.obj file2.obj -o xr.out
```

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).

7.4.2 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 *TMS470R1x Optimizing C/C++ 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/C++ compiler.

- ☐ The **-c** option tells the linker to autoinitialize variables at run time.
- ☐ The **-cr** option tells the linker to autoinitialize variables at load time.

For more information, see section 7.17, *Linking C/C++ Code*, on page 7-83, and section 7.17.6, *The -c and -cr Linker Options*, on page 7-86.

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.

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 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/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`.

```
cl470 -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 or initializes uninitialized sections when they are combined with initialized sections. This allows you to initialize memory areas during link time without reassembling a source file. The format 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.

```
c1470 -z -f 0ABCDABCDh 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 format 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`.

```
c1470 -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:

```
cl470 -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 2K bytes.

For more information, see section 7.17, *Linking C/C++ Code*, on page 7-83.

7.4.11 Alter the Library Search Algorithm (**-I** Option, **-I** Option, and **C_DIR** Environment Variable)

Usually, when you want to specify a library as linker input, you simply enter the filename; the linker looks for the library 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:

```
cl470 -z file1.obj object.lib
```

If you want to use a library that is not in the current directory, use the `-I` (lowercase L) linker option. The syntax for this option is:

```
-I [pathname] filename
```

The *filename* is the name of an archive library; the space between `-I` and the filename is optional.

The `-I` 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 7.8.6, *Allocating an Archive Member to an Output Section* on page 7-43.

You can augment the linker's directory search algorithm by using the `-I` linker option or the `C_DIR` environment variable (described in section 7.4.11.2). The linker searches for object libraries in the following order:

- 1) It searches directories named with the `-I` linker option. The `-I` option must appear before the `-I` option on the command line or in a command file.
- 2) It searches directories named with `C_DIR`.

- 3) If C_DIR is not set, it searches directories named with the assembler's 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 object libraries. 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 **-I** 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.

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	c1470 -z f1.obj f2.obj -I/ld -I/ld2 -lr.lib -llib2.lib
Windows	\ld and \ld2	c1470 -z f1.obj f2.obj -I\ld -I\ld2 -lr.lib -llib2.lib

7.4.11.2 Name an Alternate Library Directory (**C_DIR** Environment Variable)

An environment variable is a system symbol that you define and assign a string to. The linker uses an environment variable named **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)	C_DIR =" <i>pathname₁;pathname₂; . . .</i> "; export C_DIR
Windows	set C_DIR= <i>pathname₁;pathname₂; . . .</i>

The *pathnames* are directories that contain input files. Use the **-l** (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.

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)	<code>/ld</code> and <code>/ld2</code>	<code>C_DIR="/ld ;/ld2"; export C_DIR cl470 -z f1.obj f2.obj -l r.lib -l lib2.lib</code>
Windows	<code>\ld</code> and <code>\ld2</code>	<code>set C_DIR=\ld;\ld2 cl470 -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 `A_DIR` to name alternate directories that contain `.copy/include` files or macro libraries. If `C_DIR` is not set, the linker searches for object libraries in the directories named with `A_DIR`. For more information, see section 7.6, *Object Libraries*, on page 7-24.

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 `.clink` assembler directive are affected by conditional linking. See page 4-31 for details on setting up conditional linking using the `.clink` 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.

- **Attributes.** This specifies up 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

- **Fill.** This specifies a fill character for the memory range.

For more information about the MEMORY directive, see section 7.7, *The MEMORY Directive*, on page 7-26.

- 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.

- **Page.** This is the memory page of the output section. All program memory is on page 0.

- **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.

For more information about the SECTIONS directive, see section 7.8, *The SECTIONS Directive*, on page 7-30.

- ☐ 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`:

```
cl470 -z file1.obj file2.obj -m map.out
```

Example 7–21 on page 7-89 shows an example of a map file.

7.4.14 Name an Output Module (`-o filename` 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`:

```
cl470 -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 and named `nosym.out`:

```
cl470 -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 TMS470 C/C++ compiler uses an uninitialized section, `.stack`, to allocate space for the user mode run-time stack. You can set the size of the `.stack` section 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:

```
cl470 -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 stack size is 2K bytes.

7.4.17 Generate Far Call Trampolines (**--trampolines Option**)

The TMS470 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.

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, on page 7-59). 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 = EXT_MEM, run = D_MEM, start(foo_start),
    size(foo_size)
        { x.obj(.text) }

    .text: {} > EXT_MEM

    .far : { -lrts.lib(.text) } > FAR_MEM
}
```

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 `rts.lib` contains a member that defines the symbol `syntab`; none of the object files being linked reference `syntab`.

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 `rts.lib` for the member that defines `symtab` and to link in the member.

```
cl470 -z -u symtab file1.obj file2.obj rts.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 on page 7-30. For more information about the default allocation actions of the linker, see section 7.12 on page 7-52.

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:

```
cl470 -z -la.lib -lb.lib -la.lib
```

or you can force the linker to do it for you:

```
cl470 -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 `rts32.lib` without providing a full replacement for `rts32.lib`. Using `-priority` and linking your new library before `rts32.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, the call 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 on page 7-26). The SECTIONS directive controls how sections are built and allocated (see section 7.8 on page 7-30).
- ☐ Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the `cl470 -z` command and follow it with the name of the command file:

```
cl470 -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:

```
cl470 -z link.cmd
```

You can place other parameters on the command line when you use a command file:

```
cl1470 -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:

```
cl1470 -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 */
{
    D_MEM:  origin = 100h    length = 0100h
    EXT_MEM: origin = 01000h length = 01000h
}

SECTIONS                   /* SECTIONS directive */
{
    .text: > EXT_MEM
    .data: > EXT_MEM
    .bss:  > D_MEM
}
```

For more information about the MEMORY directive, see section 7.7, *The MEMORY Directive*, on page 7-26. For more information about the SECTIONS directive, see section 7.8, *The SECTIONS Directive*, on page 7-30.

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	l (lowercase L)	ORIGIN
ALIGN	len	page
attr	length	PAGE
ATTR	LENGTH	range
block	load	run
BLOCK	LOAD	RUN
COPY	MEMORY	SECTIONS
DSECT	NOLOAD	spare
f	o	type
fill	org	TYPE
group	origin	UNION
GROUP		

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*, on page 3-16) or the scheme used for integer constants in C syntax.

Examples:

Format	Decimal	Octal	Hexadecimal
Assembler format	32	40q	20h
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, the `-x` option can be used to reread libraries until no more references can be resolved (see section 7.4.20, *Exhaustively Read and Search Libraries(-x and -priority Options)*, on page 7-19). 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:

```
cl1470 -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:

```
cl1470 -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)*, on page 7-18.) The next example creates an undefined symbol `route1` in the linker's global symbol table:

```
cl1470 -z -u route1 libc.lib
```

If any member of `libc.lib` defines `route1`, the linker includes that member.

It is not possible to control the allocation of individual library members; members are allocated according to the `SECTIONS` directive default allocation algorithm. For more information, see section 7.8, *The SECTIONS Directive*, on page 7-30 .

Section 7.4.11, *Alter the Library Search Algorithm (-I Option, -I Option, C_DIR Environment Variable)*, on page 7-11, describes methods for specifying directories that contain object libraries.

7.7 The MEMORY Directive

The linker determines where output sections should be allocated in memory; it must have a model of target memory to accomplish this task. 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 TMS470 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*, on page 2-11 and section 2.4, *Relocation*, on page 2-13.

7.7.1 Default Memory Model

The assembler inserts a field in the output file's header, identifying the TMS470R1x device. The linker reads this information from the object file's header. If you do not use the MEMORY directive, the linker uses a default memory model specific to the named device. For more information about the default memory model, see section 7.12, *Default Allocation Algorithm*, on page 7-52.

7.7.2 MEMORY Directive Syntax

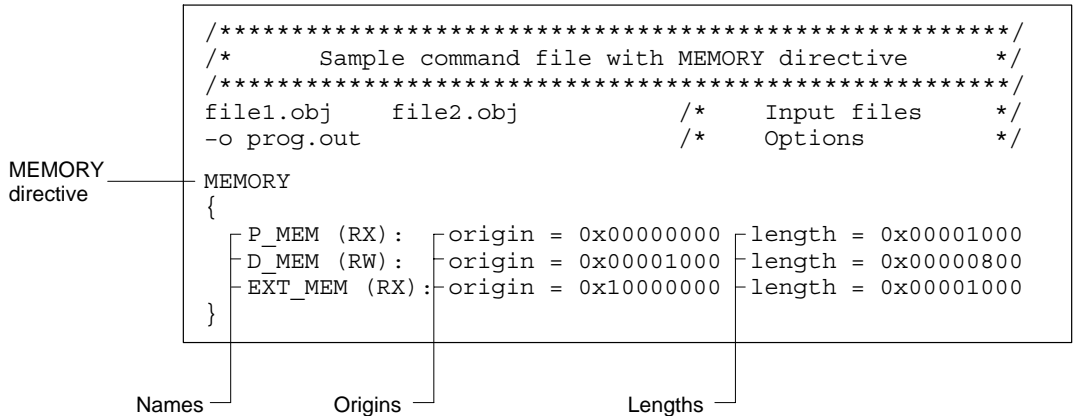
The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each range has several characteristics:

- ☐ 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 1000h, and 4K bytes of slow external memory at address 0x1000 0000h.

Example 7–3. The MEMORY Directive



The general syntax for the MEMORY directive is:

```

MEMORY
{
    name [(attr)] : origin = constant, length = constant [, fill = constant]
    .
    name [(attr)] : origin = constant, length = constant [, fill = constant]
}

```

name 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.

<i>attr</i>	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: 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
origin	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.
length	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.
fill	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.

Note: Filling Memory Ranges

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 0x0FFFFFFF:

```
MEMORY
{
    RFILE (RW) : o = 0x0020h, l = 0x1000, f = 0x0FFFFFFF
}
```

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 P_MEM and allocate the .bss section into the area name D_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*, on page 2-11, section 2.4, *Relocation*, on page 2-13, and section 2.2.4, *Subsections*, on page 2-7. 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*, on page 7-52, 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. 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)*, on page 7-51.

- ❑ **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*, on page 7-62.

Example 7–4 shows a SECTIONS directive in a sample linker command file.

Example 7–4. The SECTIONS Directive

The diagram illustrates the SECTIONS directive and its section specifications. On the left, two labels with leader lines point to the corresponding parts of the code block:

- SECTIONS directive**: Points to the `SECTIONS` keyword.
- Section specifications**: Points to the list of section definitions within the curly braces.

```

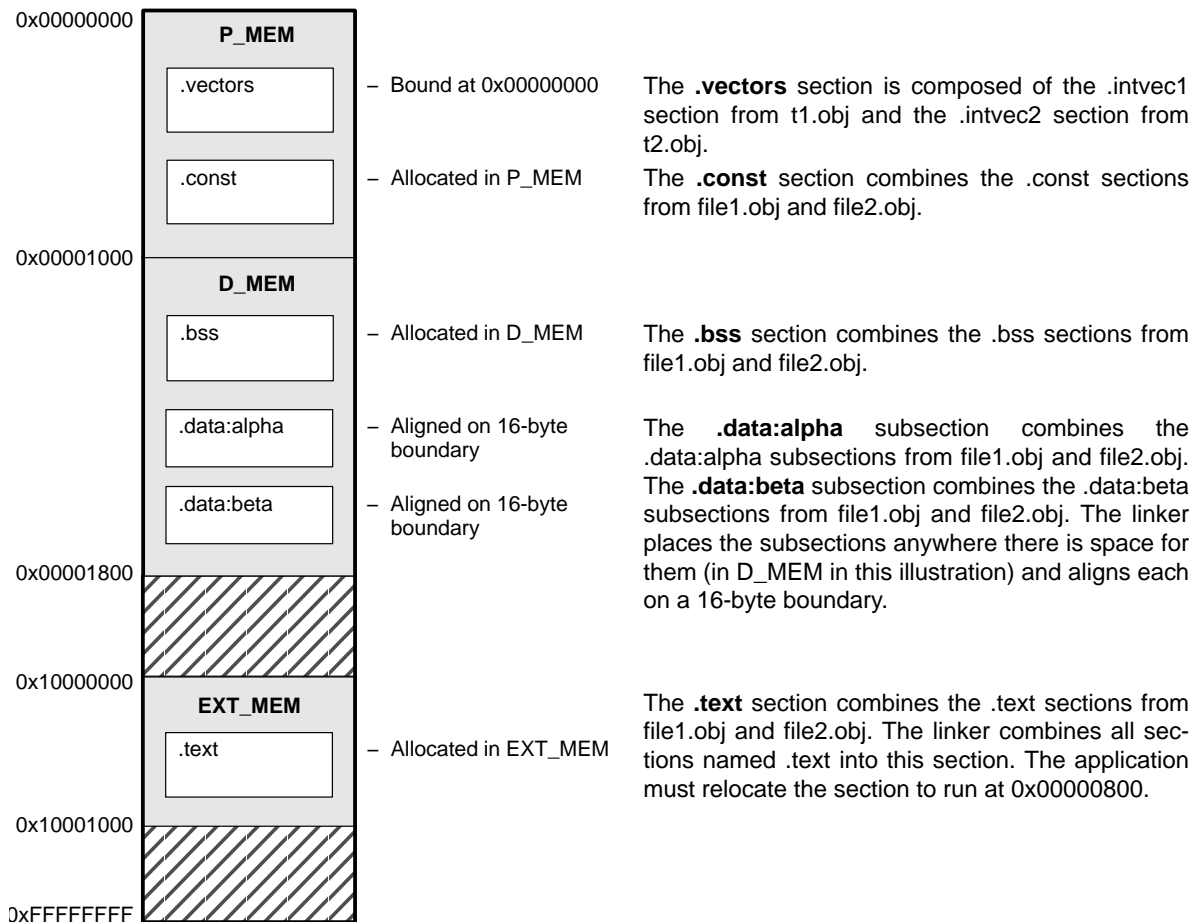
/*****
/*      Sample command file with SECTIONS directive      */
/*****
file1.obj      file2.obj          /*  Input files  */
-o prog.out          /*  Options    */

SECTIONS
{
    .text:          load = EXT_MEM, run = 00000800h
    .const:         load = P_MEM
    .bss:           load = D_MEM
    .vectors:       load = 0x00000000
    {
        t1.obj(.intvec1)
        t2.obj(.intvec2)
    }
    .data:alpha:    align = 16
    .data:beta:     align = 16
}

```

Figure 7–2 shows the five 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*, on page 7-44.

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 allocations are separate, all parameters following the keyword `LOAD` apply to load allocation, and those following the keyword `RUN` apply to run allocation. Possible 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 `EXT_MEM`) or attributes.

```
.text: load > EXT_MEM
```

Alignment uses the `align` keyword to specify that the section should start on an address boundary.

```
.text: align = 0x80
```

Blocking uses the `block` keyword to specify that the section must fit between two address boundaries: if the section is too big, it will start on an address boundary.

```
.text: block(0x80)
```

For the load (usually the only) allocation, you may simply use a greater-than sign and omit the load keyword:

```
.text: > EXT_MEM            .text: {...} > EXT_MEM  
.text: > 0x1000
```

If more than one parameter is used, you can string them together as follows:

```
.text: > EXT_MEM align 16
```

Or, if you prefer, use parentheses for readability:

```
.text: load = (EXT_MEM align(16))
```

The following sections describe these allocation parameters.

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*, on page 7-38.

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 1000h. 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.

Note: Binding is Incompatible with Alignment and Named Memory

You cannot bind a section to an address if you use alignment or named memory. If you try to do so, 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*, on page 7-26). This example names ranges and links sections into them:

```
MEMORY
{
    EXT_MEM (RIX) : origin = 0x00000000, length = 0x00001000
    D_MEM (RWIX)  : origin = 0x30000000, length = 0x00000300
}

SECTIONS
{
    .text : > EXT_MEM
    .data : > D_MEM ALIGN(128)
    .bss  : > D_MEM
```

In this example, the linker places .text into the area called EXT_MEM. The .data and .bss output sections are allocated into D_MEM. You can align a section within a named memory range; the .data section is aligned on a 128-byte boundary within the D_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 `EXT_MEM` or `D_MEM` area, because both areas have the `X` attribute. The `.data` section can also go into either `EXT_MEM` or `D_MEM`, because both areas have the `R` and `I` attributes. The `.bss` output section, however, must go into the `D_MEM` area, because only `D_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 that 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 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, the following code allocates `.text` so that it falls on a 128-byte boundary:

```
.text: load = align(128)
```

Blocking is a weaker form of alignment that allocates a section anywhere *within* a block of size `n`. If the section is larger than the block size, the section begins on that boundary. The specified block size must be a power of 2. For example, the following code allocates `.bss` so that the section either is contained in a single 128K-byte page or begins on that boundary:

```
bss: load = block(0x80)
```

You can use alignment or blocking alone or in conjunction with a memory area, but alignment and blocking cannot be used together.

7.8.2.4 Alignment With Padding

As with `align`, 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 `palign` keyword. In addition, `palign` ensures that the size of the section is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

For example, the following code lines allocate `.text` on a 4-byte boundary within the `PMEM` area. The `.text` section size is guaranteed to be a multiple of 4 bytes. Both statements are equivalent:

```
.text: palign(4) { } > PMEM  
.text: palign = 4 { } > PMEM
```

If the linker adds padding to an initialized output section then the padding space is also initialized. By default, padding space is filled with a value of 0 (zero). However, if a fill value is specified for the output section then any padding for the section is also filled with that fill value.

For example, consider the following section specification:

```
.mytext: palign(8), fill = 0xffffffff {} > PMEM
```

In this example, the length of the `.mytext` section is 17 bytes before the `palign` operator is applied. The contents of `.mytext` are as follows:

```
addr content
----
0000 0x12345678
0004 0x12345678
0008 0x12345678
000c 0x12345678
0010 0x12
```

After the `palign` operator is applied, the length of `.mytext` is 24 bytes, and its contents are as follows:

```
addr content
----
0000 0x12345678
0004 0x12345678
0008 0x12345678
000c 0x12345678
0010 0x12ffffff
0014 0xffffffff
```

The size of `.mytext` has been bumped to a multiple of 8 bytes and the padding created by the linker has been filled with `0xff`.

The fill value specified in the linker command file is interpreted as a 32-bit constant, so if you specify this code:

```
.mytext: palign(8), fill = 0xff {} > PMEM
```

The fill value assumed by the linker is `0x000000ff`, and `.mytext` will then have the following contents:

```
addr content
----
0000 0x12345678
0004 0x12345678
0008 0x12345678
000c 0x12345678
0010 0x120000ff
0014 0x000000ff
```

If the `palign` operator is applied to an uninitialized section, then the size of the section is bumped to the appropriate boundary, as needed, but any padding created is *not* initialized.

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 7–5 shows the most common type of section specification; note that no input sections are listed.

Example 7–5. The Most Common Method of Specifying Section Contents

```
SECTIONS
{
    .text:
    .data:
    .bss:
}
```

In Example 7–5 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 7–5 are actually a shorthand method for the following:

```
SECTIONS
{
    .text: { *(.text) }
    .data: { *(.data) }
    .bss:  { *(.bss) }
}
```

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 preceding concepts:

```
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 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.

7.8.5 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 output sections should not be split:

- ☐ 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
- ☐ An output section with separate load and run allocations. The code that copies the output section from its load-time allocation to its run-time location cannot accommodate a split in the output section.
- ☐ 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.

If you use the >> operator on any of these sections, the linker issues a warning and ignores the operator.

7.8.6 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[, member2[, ...]]*> [*(input 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 above example 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.

The *-l* option (which normally implies a library path search be made for the named file following the option) listed before each library is optional when listing specific archive members inside < >. Using < > implies that you are referring to a library.

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 = EXT_MEM, run = D_MEM
```

Use the *load* keyword for the load address and the *run* keyword for the run address.

See section 2.5, *Run-Time Relocation*, on page 2-15, 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. All 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*, on page 7-48.)

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 = EXT_MEM, align = 32, run = D_MEM
```

(align applies only to load)

```
.data: load = (EXT_MEM align 32), run = D_MEM
```

(identical to previous example)

```
.data: run      = D_MEM, align 32,
      load      = align 16
```

(align 32 in D_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 = 0x00001000, run = D_MEM
```

A warning is issued, load is ignored, and space is allocated in D_MEM. All of the following examples have the same effect. The .bss section is allocated in D_MEM.

```
.bss: load = D_MEM
.bss: run = D_MEM
.bss: > D_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. For more information on the .label directive, see page 4-50.

Example 7-6 shows the use of the .label directive. Figure 7-3 illustrates the run-time execution of Example 7-6.

Example 7–6. Copying a Section From EXT_MEM to P_MEM**(a) Assembly language file**

```

;-----
;   define a section to be copied from EXT_MEM to P_MEM
;-----
        .sect  ".fir"
        .label fir_src      ; load address of section
fir:    <code here>         ; run address of section
        .label fir_end     ; code for section
        .label fir_end     ; load address of section end

;-----
;   copy .fir section from EXT_MEM to P_MEM
;-----
        .text

        LDR    r4, fir_s    ; get fir load address start
        LDR    r5, fir_e    ; get fir load address stop
        LDR    r3, fir_a    ; get fir run address
$1:     CMP    r4, r5
        LDRCC  r0, [r4], #4 ; copy fir routine to its
                               ; run address
        STRCC  r0, [r3], #4
        BCC    $1

;-----
;   jump to fir routine, now in P_MEM
;-----
        B      fir

fir_a    .word  fir
fir_s    .word  fir_start
fir_e    .word  fir_end

```

(b) Linker command file

```

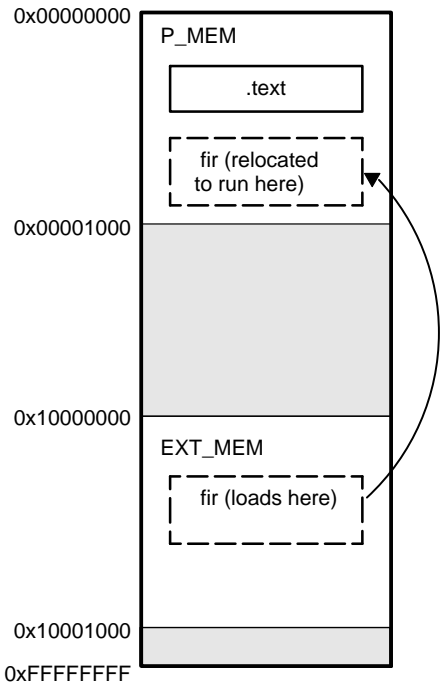
/*****
/*   PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE   */
*****/

MEMORY
{
    P_MEM      :  origin = 0x00001000, length = 0x00001000
    EXT_MEM    :  origin = 0x10000000, length = 0x00001000
}

SECTIONS
{
    .text: load = P_MEM
    .fir:  load = EXT_MEM, run P_MEM
}

```

Figure 7-3. Run-Time Execution of Example 7-6



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 section, subsection, 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 7–7, the .bss sections from file1.obj and file2.obj are allocated at the same address in D_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 7–7. The UNION Statement

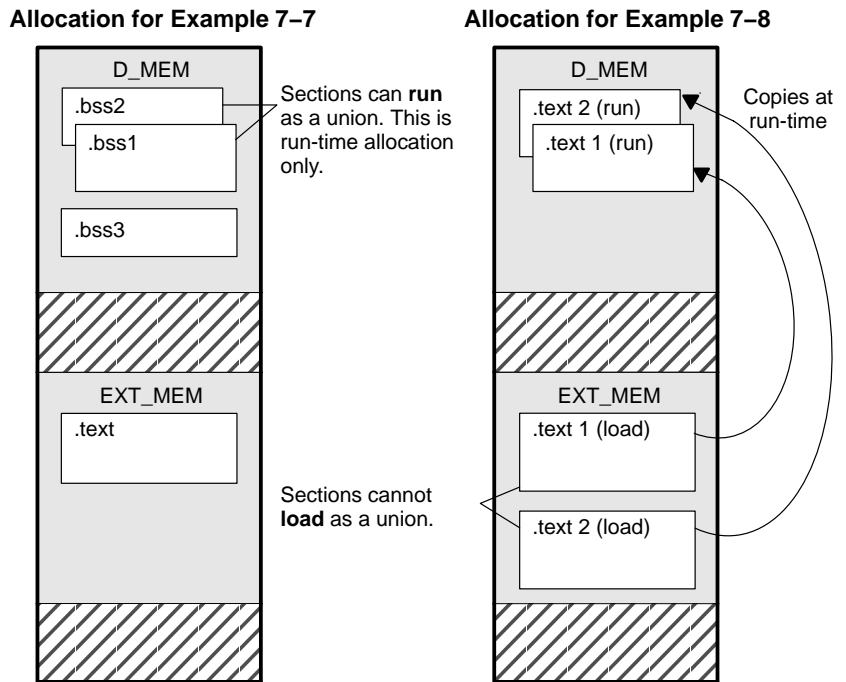
```
SECTIONS
{
    .text: load = EXT_MEM
    UNION: run = D_MEM
    {
        .bss:part1: { file1.obj(.bss) }
        .bss:part2: { file2.obj(.bss) }
    }
    .bss:part3: run = D_MEM { globals.obj(.bss) }
```

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. For example:

Example 7–8. Separate Load Addresses for UNION Sections

```
UNION: run = D_MEM
{
    .text:part1: load = EXT_MEM, { file1.obj(.text) }
    .text:part2: load = EXT_MEM, { file2.obj(.text) }
}
```


Figure 7-4. Memory Allocation Shown in Example 7-7 and Example 7-8



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 the 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 7–9. 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.

Note: You Cannot Specify Addresses for Sections Within a GROUP

When you use the GROUP option, binding, alignment, or allocation into named memory can be specified *for the group only*. You cannot use binding, named memory, or alignment for sections *within* a group.

7.11 Special Section Types (DSECT, COPY, and NOLOAD)

You can assign three special type designations 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 (enclosed in parentheses) after the section definition. For example:

```
SECTIONS
{
    sec1 :    {f1.obj}    (DSECT)    0x00002000
    sec2 :    {f2.obj}    (COPY)     0x00004000
    sec3 :    {f3.obj}    (NOLOAD)   0x00006000
}
```

- ❑ 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 of the symbols are relocated as though the sections were linked at address 2000h. 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 TMS470R1x 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 7–10 were specified.

Example 7–10. Default Allocation for TMS470R1x Devices

```
MEMORY
{
    RAM      : origin = 0x00000000, length = 0x80000
}

SECTIONS
{
    .text : > RAM
    .const: > RAM
    .data : > RAM
    .bss  : > RAM
    .cinit: > 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.

7.12.1 How the Allocation Algorithm Creates Output Sections

An output section can be formed in one of two ways:

- Method 1** An output section can be formed as the result of a SECTIONS directive definition.
- Method 2** An output section can be formed by combining input sections with the same names 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*, on page 7-30, 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 file 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)*, on page 7-19) 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 7-10. (See section 7.7, *The MEMORY Directive*, on page 7-26, 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/C++ language:

```
symbol    = expression;  assigns the value of expression to symbol
symbol    += expression;  adds the value of expression to symbol
symbol    -= expression;  subtracts the value of expression from symbol
symbol    *= expression;  multiplies symbol by expression
symbol    /= expression;  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 of Expressions*, on page 7-55. 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*, on page 7-30.)

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 page 4-44), 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*, on page 7-62.

7.13.3 Assignment Expressions

These rules apply to linker expressions:

- ☐ Expressions can contain global symbols, constants, and the C/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 zero 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 page 4-44). 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.
- __SYSTEMEM_SIZE** is assigned the size of the `.systemem` 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 7–6) 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 7–6.

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:

LOAD_START(sym) START(sym)	Defines <i>sym</i> with the load-time start address of related allocation unit
LOAD_END(sym) END(sym)	Defines <i>sym</i> with the load-time end address of related allocation unit
LOAD_SIZE(sym) SIZE(sym)	Defines <i>sym</i> with the load-time size of related allocation unit
RUN_START(sym)	Defines <i>sym</i> with the run-time start address of related allocation unit
RUN_END(sym)	Defines <i>sym</i> with the run-time end address of related allocation unit
RUN_SIZE(sym)	Defines <i>sym</i> with the run-time size of related allocation unit

Note: Linker Command File Operator Equivalencies

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. The following text describes how the linker handles such holes and how you can fill holes (and uninitialized sections) with a value.

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 page 4-28) and sections defined with the `.usect` directive (see page 4-76) 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. There is no way to fill or initialize the space between output sections.

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*, on page 7-54.

The following example uses assignment statements to create holes in output sections:

```
SECTIONS
{
    outsect:
    {
        file1.obj(.text)
        . += 100h;          /* Create a hole with size 100h    */
        file2.obj(.text)
        . = align(16);      /* Create a hole to align the SPC */
        file3.obj(.text)
    }
}
```

The output section outsect is built as follows:

- ☐ The .text section from file1.obj is linked in.
- ☐ The linker creates a 256-byte hole.
- ☐ The .text section from file2.obj is linked in after the hole.
- ☐ The linker creates another hole by aligning the SPC on a 16-byte boundary.
- ☐ 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 `.` 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: { . += 100h; } /* Hole at the beginning */
.data: {
        *(.data)
        . += 100h; } /* 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:

```
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 */
    {
        . += 10h;                    /* 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)*, on page 7-10). For example, suppose the command file `link.cmd` contains the following `SECTIONS` directive:

```
SECTIONS
{
    .text: { .= 100; }      /* Create a 100-byte hole */
}
```

Now invoke the linker with the `-f` option:

```
cl470 -z -f 0xFFFFFFFF link.cmd
```

This fills the hole with `0xFFFFFFFF`.

- 4) 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 */
}
```

Note: Filling Sections

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 <code>.flashcode</code> section
<code>_flash_code_rn_start</code>	Run address of <code>.flashcode</code> section
<code>_flash_code_size</code>	Size of <code>.flashcode</code> 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*, on page 7-59.

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 7–11:

Example 7–11. 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 7–11 can be written as shown in Example 7–12:

Example 7–12. 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 7–12 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 7–13:

Example 7–13. 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 7–14 shows the TMS320C6000 copy table header file.

Example 7–14. TMS470 cpy_tbl.h File

```
/* **** */
/* cpy_tbl.h                                     */
/* **** */
/* Copyright (c) 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 page id
- ☐ The run page id
- ☐ The load address
- ☐ The run address
- ☐ The size

The linker collects all `COPY_RECORD`s 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_RECORD`s in the table, and the array of `COPY_RECORD`s 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 page id of .first>,
      <run page id of .first>,
      { <load address of .first>,
        <run address of .first>,
        <size of .first> } },
    { <load page id of .extra>,
      <run page id of .extra>,
      { <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 7–14 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 7–15.

Example 7–15. Run-Time-Support cpy_tbl.c File

```

/*****
/* cpy_tbl.c
/*
/* Copyright (c) 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 int i;
    for (i = 0; i < tp->num_recs; i++)
    {
        COPY_RECORD crp = tp->recs[i];
        unsigned int cpy_type = 0;
        unsigned int j;

        if (crp->src_pgid) cpy_type += 2;
        if (crp->dst_pgid) cpy_type += 1;

        for (j = 0; j < crp->size; j++)
        {
            switch (cpy_type)
            {
                case 3: ddcopy(crp->src_addr + j, crp->dst_addr + j); break;
                case 2: dpcopy(crp->src_addr + j, crp->dst_addr + j); break;
                case 1: pdcopy(crp->src_addr + j, crp->dst_addr + j); break;
                case 0: ppcopy(crp->src_addr + j, crp->dst_addr + j); break;
            }
        }
    }
}

```

The load (or source) page id and the run (or destination) page id are used to choose which low-level copy routine is called to move a word of data from the load location to the run location. A page id of 0 indicates that the specified address is in program memory, and a page id of 1 indicates that the address is in data memory. The hardware provides special instructions, PREAD and PWRITE, to move code/data into and out of program memory.

The source for the low-level copy routines is included in another run-time-support source file called cpy_utils.asm as shown in Example 7–16.

Example 7–16. The cpy_utils.asm File

```

;*****
; CPY_UTILS.ASM
;
; Copy utility functions to perform 4 different types of copy,
; data -> data, data -> program, program -> data, and
; program -> program.
;
; source address always arrives in the accumulator 'ACC'
; destination address arrives on the stack '*-SP[2]'
;
; We will use XAR6 and XAR7 as temporary registers in
; these routines.
;
;*****

    .if      .TMS470
    .asg     LRETR, RETURN
    .asg     MOVL,  MOVE_L
    .else
    .asg     RET,   RETURN
    .asg     MOV,   MOVE_L
    .endif

    .sect    ".text"
    .global  _ddcopy
;*****
; Data Memory -> Data Memory
;
; Use the accumulator to hold the data being moved between the
; load and the store.
;*****
_ddcopy:
    ; Load word at Source Address into Accumulator
    MOVL     XAR6,ACC
    MOV      AL,*+XAR6[0]

    ; Store word in Accumulator at Dest Address
    MOVE_L   XAR6,*-SP[4]
    MOV      *+XAR6[0],AL

    RETURN

    .sect    ".text"
    .global  _dpcopy
;*****
; Data Memory -> Program Memory
;
; Transfer the dest address into XAR7 so that we can use a PWRITE
; instruction to store the data into program memory.
;*****
_dpcopy:
    ; Load Dest address into XAR7
    MOVE_L   XAR7,*-SP[4]

```

Example 7–16. The cpy_utils.asm File (Continued)

```

        ; Write data to Program Memory
        MOVL    XAR6,ACC
        PWRITE  *XAR7,*XAR6
        RETURN

        .sect   ".text"
        .global _pdcopy
;*****
; Program Memory -> Data Memory
;
; Put source address into XAR7 so we can use a PREAD instruction to
; get the data from program memory.
;*****
_pdcopy:
        ; Load Source address into XAR7
        MOVL    XAR7,ACC

        ; Read from Program Memory into Dest Address (data memory)
        MOVE_L  XAR6,*-SP[4]
        PREAD   *XAR6,*XAR7

        RETURN

        .sect   ".text"
        .global _ppcopy
;*****
; Program Memory -> Program Memory
;
; First read the data from program memory into the accumulator.
; Then store it into the destination location (also in program
; memory). We use XAR7 to hold first the source address and then
; the dest address so that we can make use of the PREAD and
; PWRITE instructions for reading/writing data from/to program
; memory.
;*****
_ppcopy:
        ; Load Source Address into XAR7
        MOVL    XAR7,ACC

        ; Read from Program Memory into Accumulator
        PREAD   @AL,*XAR7

        ; Load Dest address into XAR7
        MOVE_L  XAR7,*-SP[4]

        ; Write Accumulator contents to Program Memory
        PWRITE  *XAR7,@AL

        RETURN

```

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 7–17 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 7–17. 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 7–17, 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 7–18. Example 7–19 illustrates a possible driver for such an application.

Example 7–18. 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 7–19. 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, *Relocation Capabilities, (-a and -r Options)*, on page 7-6.)
- ☐ 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)*, on page 7-15.)
- ☐ 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 wish 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)*, on page 7-10.)
- ☐ 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)*, on page 7-9.)

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`.

```
cl470 -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`.

```
cl470 -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`:

```
cl470 -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:

```
cl470 -z -c -o prog.out prog1.obj prog2.obj ... rts32.lib
```

The `-c` option tells the linker to use special conventions that are defined by the C environment. The archive libraries `rts16.lib` and `rts32.lib` contains C run-time-support functions.

For more information about C, including the run-time environment and run-time-support functions, see the *TMS470R1x Optimizing C/C++ Compiler User's Guide*.

7.17.1 Run-Time Initialization

All C programs must be linked with an object module called `boot.obj`. When a program begins running, it executes `boot.obj` first. The `boot.obj` module contains code and data for initializing the run-time environment. The module performs the following tasks:

- ☐ Changes from system mode to user mode
- ☐ Sets up the user mode stack
- ☐ Processes the run-time initialization table and autoinitializes global variables when the linker is invoked with the `-c` option
- ☐ 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 *TMS470R1x 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 `rts32.lib` or `rts16.lib` 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 .sysmem Sections

The C/C++ language uses two uninitialized sections called .sysmem 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 constant immediately after the option. The default size for both, if the options are not used, is 2K bytes.

See section 7.4.10, *Define .heap Size (-heap size Option)*, on page 7-11, and section 7.4.16, *Define .stack Size (-stack size Option)*, on page 7-16 for more information on setting stack sizes.

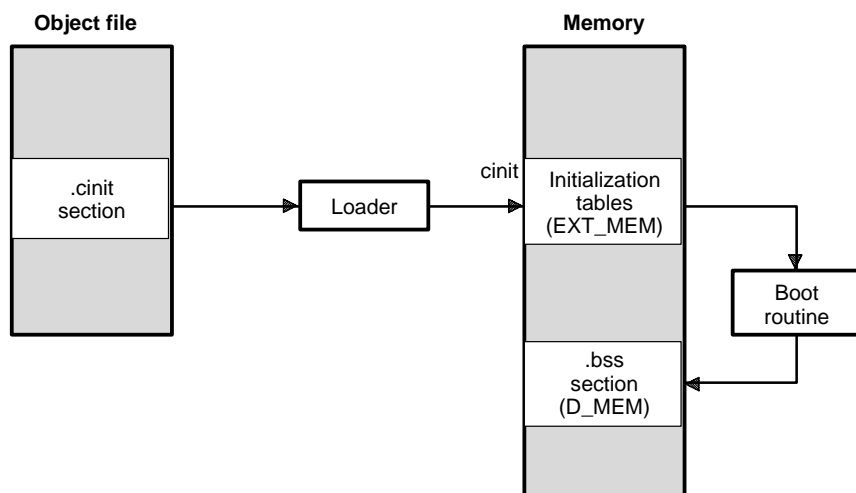
7.17.4 Autoinitialization of Variables at Run Time

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 defined 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.

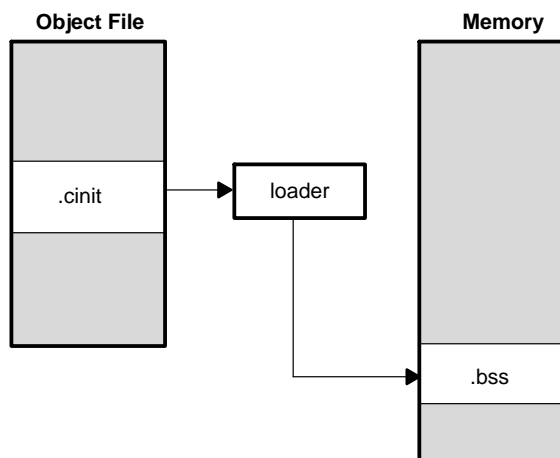
When you use the `-cr` linker option, the linker sets the `STYP_COPY` bit in the `.cinit` section's header. This tells the loader not to load the `.cinit` section into memory. (The `.cinit` section occupies no space in the memory map.) The linker also sets the `cinit` symbol to `-1` (normally, `cinit` 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 (which is not part of the compiler package) must be able to perform the following tasks to use autoinitialization 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 autoinitialization of variables at load time.

Figure 7–6. Autoinitialization 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 autoinitialization at run time or the loader autoinitialization 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`. The symbol `SETUP` is the program entry point.

Assume that target memory has the following configuration:

Program Memory

Address Range	Contents
0x00000000 to 0x00001000	EXT_MEM
0x00001000 to 0x00002000	D_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 `EXT_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 `D_MEM` and preinitialized to `0x00000100`.
- ☐ Another `.bss` section in `ctrl.obj` must be linked into `D_MEM`.

Example 7–20 shows the linker command file for this example. Example 7–21 shows the map file.

Example 7–20. 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 file */

/*****
***                               Specify the Input Files                               ***
*****/
demo.obj
ctrl.obj
tables.obj

/*****
***                               Specify the Memory Configurations                       ***
*****/
MEMORY
{
    P_MEM   : org = 0x00000000    len = 0x00001000 /* PROGRAM MEMORY (ROM) */
    D_MEM   : org = 0x00001000    len = 0x00001000 /* DATA MEMORY (RAM) */
    EEPROM  : org = 0x08000000    len = 0x00000400 /* COEFFICIENTS (EEPROM) */
}

/*****
/*                               Specify the Output Sections                          */
*****/
SECTIONS
{
    .text      : {} > P_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 sections for ctrl variables */
    {
        ctrl.obj(.bss)
    } = 0x00000100 > D_MEM        /* Fill with 0x100 and link into RAM */
    .bss       : {} > D_MEM        /* Link remaining .bss sections into RAM */
}

/*****
***                               End of Command File                               ***
*****/

```

Invoke the linker with the following command:

```
cl1470 -z demo.cmd
```

This creates the map file shown in Example 7–21 and an output file called demo.out that can be run on a TMS470R1x.

Example 7-21. Output Map File, demo.map

```

OUTPUT FILE NAME:  <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address: 000000d4

MEMORY CONFIGURATION

      name      origin      length      attributes      fill
      -----      -
P_MEM      00000000      000001000      RWIX
D_MEM      00001000      000001000      RWIX
EEPROM      08000000      000000400      RWIX

SECTION ALLOCATION MAP

      output      page      origin      length      attributes/
      section      -----      -
      .text      0      00000020      00000138
                        00000020      000000a0      ctrl.obj (.text)
                        000000c0      00000000      tables.obj (.text)
                        000000c0      00000098      demo.obj (.text)
      .intvecs      0      00000000      00000020
                        00000000      00000020      tables.obj (.intvecs)
      .data      0      08000000      00000400
                        08000000      00000168      tables.obj (.data)
                        08000168      00000298      --HOLE-- [fill = ff00ff00]
                        08000400      00000000      ctrl.obj (.data)
                        08000400      00000000      demo.obj (.data)
      ctrl_var      0      00001000      00000500
                        00001000      00000500      ctrl.obj (.bss) [fill = 00000100]
      .bss      0      00001500      00000100      UNINITIALIZED
                        00001500      00000100      demo.obj (.bss)
                        00001600      00000000      tables.obj (.bss)

GLOBAL SYMBOLS

      address      name      address      name
      -----      -
00001500 .bss      00000020 clear
08000000 .data      00000020 .text
00000020 .text      000000b8 set
000000d4 SETUP      000000c0 x42
00000020 clear      000000d4 SETUP
08000400 edata      00000158 etext
00001600 end      00001500 .bss
00000158 etext      00001600 end
000000b8 set      08000000 .data
000000c0 x42      08000400 edata

[10 symbols]

```

Absolute Lister Description

The TMS470R1x 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 is a tedious process requiring many operations; however, the absolute lister utility performs these operations automatically.

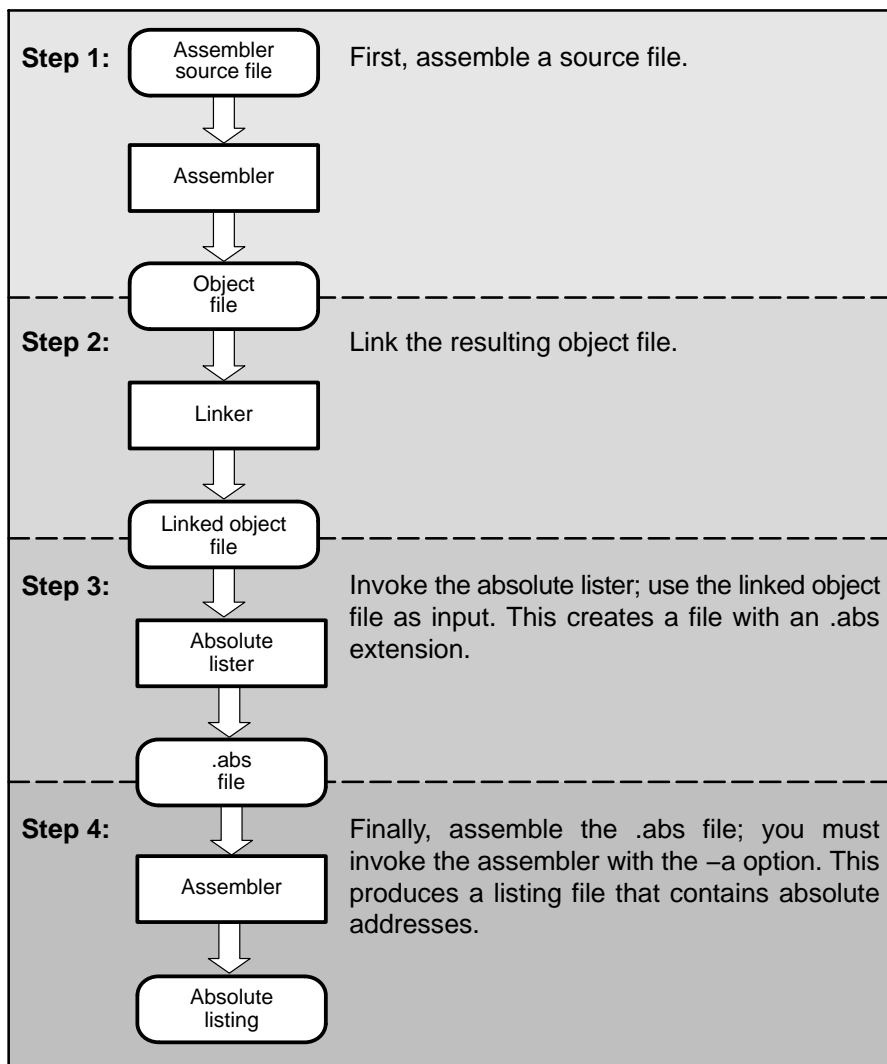
This chapter explains how to invoke the absolute lister to obtain a listing of the absolute addresses of an object file.

Topic	Page
8.1 Producing an Absolute Listing	8-2
8.2 Invoking the Absolute Lister	8-3
8.3 Absolute Lister Example	8-5

8.1 Producing an Absolute Listing

Figure 8–1 illustrates the steps required to produce an absolute listing.

Figure 8–1. *The Absolute Lister in the TMS470R1x Software Development Flow*



8.2 Invoking the Absolute Lister

The syntax for invoking the absolute lister is:

abs470 [*options*] *input file*

abs470 is the command that invokes the absolute lister.

options 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:

-e enables you to change abs470's default naming conventions for filename extensions on assembly files, C source files, and C header files. The three -e options are listed below.

☐ **-ea** [*asmext*] for assembly files (default is .asm)

☐ **-ec** [*cext*] for C source files (default is .c)

☐ **-eh** [*hext*] for C header files (default is .h)

The . in the extensions and the space between the option and the extension are optional.

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.

-q (quiet) suppresses the banner and all progress information.

input file 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 -a assembler option as follows to create the absolute listing:

```
asm470 -a filename.abs
```

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 debugging set; this generates the `hello.s` assembly file. The file `hello.csr` also includes `hello.hsr`. Assuming the executable file created is called `hello.out`, the following command generates the proper `.abs` file:

```
abs470 -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`.

8.3 Absolute Lister Example

This example uses three source files. The module1.asm and module2.asm files both include the file globals.def.

module1.asm

```
.text
.bss    dflag, 1
.bss    array, 100
dflag_a .word dflag
array_a .word array
offst_a .word offst
.copy   globals.def

LDR     r4, array_a
LDR     r5, offst_a
LDR     r3, dflag_a
LDR     r0, [r4, r5]
STR     r0, [r3]
```

module2.asm

```
.text
.bss    offst, 1
offst_a .word offst
.copy   globals.def

LDR     r4, offst_a
STR     r0, [r4]
```

globals.def

```
.global array
.global offst
.global dflag
```

The following steps create absolute listings for the files module1.asm and module2.asm:

Step 1: First, assemble module1.asm and module2.asm:

```
asm470 module1
asm470 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:

```
/*
File bttest.cmd -- COFF linker command file
for linking TMS470R1x modules
*/
-o bttest.out          /* Name the output file */
-m bttest.map          /* Create an output map */

/*
Specify the Input Files
*/
module1.obj
module2.obj

/*
Specify the Memory Configurations
*/
MEMORY
{
    P_MEM :   org = 0x00000000   len = 0x00001000   /* Program Memory (ROM) */
    D_MEM :   org = 0x00001000   len = 0x00001000   /* Data Memory (RAM) */
}

/*
Specify the Sections Allocation into Memory
*/
SECTIONS
{
    .data:   >D_MEM
    .text:   >P_MEM
    .bss:    >D_MEM
}
```

Invoke the linker:

lnk470 bttest.cmd

This 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:

```
abs470 bttest.out
```

This creates two files called module1.abs and module2.abs:

module1.abs:

```
.nolist
array      .setsym      000001001h
dflag      .setsym      000001000h
offst      .setsym      000001068h
.data      .setsym      000001000h
edata      .setsym      000001000h
.text      .setsym      000000000h
etext      .setsym      00000002ch
.bss       .setsym      000001000h
end        .setsym      00000106ch
           .setsect     ".text",000000000h
           .setsect     ".data",000001000h
           .setsect     ".bss",000001000h
           .list
           .text
           .copy        "module1.asm"
```

module2.abs:

```
.nolist
array      .setsym      000001001h
dflag      .setsym      000001000h
offst      .setsym      000001068h
.data      .setsym      000001000h
edata      .setsym      000001000h
.text      .setsym      000000000h
etext      .setsym      00000002ch
.bss       .setsym      000001000h
end        .setsym      00000106ch
           .setsect     ".text",000000020h
           .setsect     ".data",000001000h
           .setsect     ".bss",000001068h
           .list
           .text
           .copy        "module2.asm"
```

These files contain the following information that the assembler needs when you invoke it in step 4:

- ☐ They contain .setsym directives, which equate values to global symbols. Both files contain global equates for the symbols *dflag*, *array*, and *offst*. The symbols *dflag* and *array* were defined in the file *module1.asm*. The symbol *offst* was defined in the file *module2.asm*.
- ☐ They contain .setsect directives, which define the absolute addresses for sections.

- They contain .copy directives, which tell the assembler which assembly language source file to include.

The .setsym and .setsect directives are not useful in normal assembly; they are useful only for creating absolute listings.

Step 4: Finally, assemble the .abs files created by the absolute lister (remember that you must use the -a option when you invoke the assembler):

```
asm470 -a module1.abs
asm470 -a module2.abs
```

This 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:

```
TMS470 COFF Assembler          Version x.xx          Tue Nov 12 17:35:49 1996
Copyright (c) 1996             Texas Instruments Incorporated

module1.abs                                PAGE      1

      15 00000000                      .text
      16                                .copy          "module1.asm"
A      1 00000000                      .text
A      2 00001000                      .bss      dflag, 1
A      3 00001001                      .bss      array, 100
A      4 00000000 00001000- dflag_a .word      dflag
A      5 00000004 00001001- array_a .word      array
A      6 00000008 00001068! offst_a .word      offst
A      7                                .copy      globals.def
B      1                                .global   array
B      2                                .global   offst
B      3                                .global   dflag
A      8
A      9 0000000c E51F4010             LDR       r4, array_a
A     10 00000010 E51F5010             LDR       r5, offst_a
A     11 00000014 E51F301C             LDR       r3, dflag_a
A     12 00000018 E7940005             LDR       r0, [r4, r5]
A     13 0000001c E5830000             STR       r0, [r3]

No Errors, No Warnings
```

module2.lst:

TMS470 COFF Assembler
Copyright (c) 1996

Version x.xx Tue Nov 12 17:35:59 1996
Texas Instruments Incorporated

module2.abs

PAGE 1

```

    15 00000020                .text
    16                .copy      "module2.asm"
A     1 00000020                .text
A     2 00001068                .bss      offst, 1
A     3 00000020 00001068- offst_a .word   offst
A     4                .copy      globals.def
B     1                .global  array
B     2                .global  offst
B     3                .global  dflag
A     5
A     6 00000024 E51F400C      LDR      r4, offst_a
A     7 00000028 E5840000      STR      r0, [r4]
```

No Errors, No Warnings

Cross-Reference Lister

The TMS470R1x cross-reference lister is a debugging tool that 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.

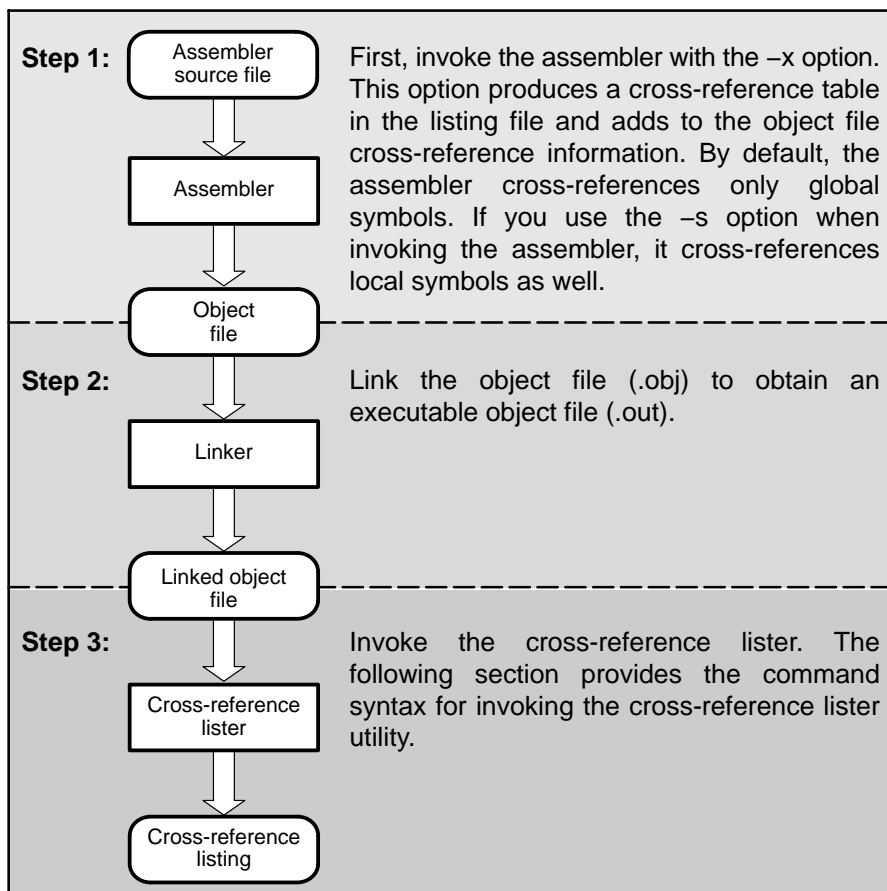
This chapter explains how to invoke the cross-reference lister to obtain a listing of symbols, their definitions, and their references in the linked source files.

Topic	Page
9.1 Producing a Cross-Reference Listing	9-2
9.2 Invoking the Cross-Reference Lister	9-3
9.3 Cross-Reference Listing Example	9-4

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 TMS470R1x 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 `-x` 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 `-s` option, local symbols are also added. Link the object files to obtain an executable file.

To invoke the cross-reference lister, enter the following:

xref470 [*options*] [*input filename* [*output filename*]]

xref470	is the command that invokes the cross-reference utility.
<i>options</i>	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. Precede each option with a hyphen (-). The cross-reference lister options are as follows: <ul style="list-style-type: none"> -l (lowercase L) specifies the number of lines per page for the output file. The format of the <code>-l</code> option is <code>-lnum</code>, where <code>num</code> is a decimal constant. For example, <code>-l30</code> 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 (quiet) suppresses the banner and all progress information.
<i>input filename</i>	is a linked object file. If you omit the input filename, the utility prompts for a filename.
<i>output filename</i>	is the name of the cross-reference listing file. If you omit the output filename, the default filename is the input filename with an <code>.xrf</code> extension.

9.3 Cross-Reference Listing Example

TMS470 XREF Utility		Version x.xx					
Copyright (c) 1996		Texas Instruments Incorporated					
File: bttest.out		Wed Nov 13 17:07:42 1996				Page: 1	
=====							
Symbol: array							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
module1.asm	EDEF	-00000001	00001001	3	1A	5	
=====							
Symbol: array_a							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
module1.asm	STAT	'00000004	00000004	5	9		
=====							
Symbol: dflag							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
module1.asm	EDEF	-00000000	00001000	2	3A	4	
=====							
Symbol: dflag_a							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
module1.asm	STAT	'00000000	00000000	4	11		
=====							
Symbol: offst							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
module1.asm	EREF	00000000	00001068		2A	6	
module2.asm	EDEF	-00000000	00001068	2	2A	3	
=====							
Symbol: offst_a							
Filename	RTYP	AsmVal	LnkVal	DefLn	RefLn	RefLn	RefLn
module1.asm	STAT	'00000008	00000008	6	10		
module2.asm	STAT	'00000000	00000020	3	6		
=====							

The terms defined below appear in the preceding cross-reference listing:

Symbol	Name of the symbol listed								
Filename	Name of the file where the symbol appears								
RTYP	The symbol's reference type in this file. The possible reference types are: <table> <tr> <td>STAT</td><td>The symbol is defined in this file and is not declared as global.</td></tr> <tr> <td>EDEF</td><td>The symbol is defined in this file and is declared as global.</td></tr> <tr> <td>EREF</td><td>The symbol is not defined in this file but is referenced as global.</td></tr> <tr> <td>UNDF</td><td>The symbol is not defined in this file and is not declared as global.</td></tr> </table>	STAT	The symbol is defined in this file and is not declared as global.	EDEF	The symbol is defined in this file and is declared as global.	EREF	The symbol is not defined in this file but is referenced as global.	UNDF	The symbol is not defined in this file and is not declared as global.
STAT	The symbol is defined in this file and is not declared as global.								
EDEF	The symbol is defined in this file and is declared as global.								
EREF	The symbol is not defined in this file but is referenced as global.								
UNDF	The symbol is not defined in this file and is not declared as global.								
AsmVal	The hexadecimal value assigned to the symbol at assembly time. A value may also be preceded by a character that describes the symbol's attributes. Table 9–1 lists these characters and names.								
LnkVal	The hexadecimal value assigned to the symbol after linking								
DefLn	The statement number where the symbol is defined								
RefLn	The line number where the symbol is referenced. If the line number is followed by an asterisk(*), then that reference may modify the contents of the object. A blank in this column indicates that the symbol was never used.								

Table 9–1. Symbol Attributes

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
=	Symbol defined in a .reg 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 human readable and XML formats.
- ☐ 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	10-2
10.2 XML Tag Index	10-3
10.3 Example XML Consumer	10-9
10.4 Invoking the Name Utility	10-16
10.5 Invoking the Strip Utility	10-17

10.1 Invoking the Object File Display Utility

The object file display utility, *ofd470*, is used to print the contents of object files (.obj), executable files (.out), and/or archive libraries (.lib) in both human readable and XML formats.

To invoke the object file display utility, enter the following:

ofd470 [options] *input filenames* [*input filenames*]

- ofd470** is the command that invokes the object file display utility.
- input* names the assembly language source file. The file name must
- filenames* contain a .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.
- ofilename** 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.

10.2 XML Tag Index

Table 10–1 describes the XML tags that are generated by the object file display utility.

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
<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

Table 10–1. XML Tag Index (Continued)

Tag Name	Context	Description
<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

Table 10–1. XML Tag Index (Continued)

Tag Name	Context	Description
<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
<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

Table 10–1. XML Tag Index (Continued)

Tag Name	Context	Description
<next_symbol>	<symbol>	Index of next symbol after multisymbol 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

Table 10–1. XML Tag Index (Continued)

Tag Name	Context	Description
<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_addr>	<symbol>	Number of machine-address-sized units in function
<size_in_bits>	<symbol>	Size of symbol (bits)
<source>	<memory>	Source register
	<register>	Source register
<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>	String table entry
	<value>	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

Table 10–1. XML Tag Index (Continued)

Tag Name	Context	Description
<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>	Attribute type
	<reloc_entry>	Type of relocation
<type_ref>	<value>	Type reference
<value>	<attribute>	Attribute value
	<reloc_entry>	Value
	<symbol>	Value
<vector>	<section>	True if this section contains a vector table (C55x only)
<version>	<compile_unit>	DWARF version
	<file_header>	Version ID; structure version of this COFF file
<virtual_addr>	<reloc_entry>	Virtual address to be relocated
	<section>	Virtual (load) address of section
<word_size>	<reloc_entry>	Number of address-sized units containing the relocation field
<xml_version>	<dwarf>	Version of the DWARF XML language
	<ti_coff>	Version of the COFF XML language

10.3 Example XML Consumer

This section provides an example of a small application that uses the XML output of ofd470 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 ofd470 from the command line as follows:

```
% ofd470 -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; }

        //-----
        // 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;

    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 Name Utility

The name utility, *nm470*, is used to print the list of names defined and referenced in a COFF object (.obj) or an executable file (.out). The value associated with the symbol and an indication of the kind of symbol is also printed.

To invoke the name utility, enter the following:

nm470 [*-options*] [*input filename*]

nm470 is the command that invokes the name utility.

input filename is a COFF object file (.obj) or an executable file (.out).

options 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:

- a** prints all symbols.
- c** also prints C_NULL symbols.
- d** also prints debug symbols.
- f** prepends file name to each symbol.
- g** prints only global symbols.
- h** shows the current help screen.
- l** produces a detailed listing of the symbol information.
- n** sorts symbols numerically rather than alphabetically.
- o<file>** outputs to the given file.
- p** causes the name utility to not sort any symbols.
- q** (quiet mode) suppresses the banner and all progress information.
- r** sorts symbols in reverse order.
- t** also prints tag information symbols.
- u** only prints undefined symbols.

10.5 Invoking the Strip Utility

The strip utility, *strip470*, is used to remove symbol table and debugging information from object and executable files.

To invoke the strip utility, enter the following:

```
strip470 [-p] input filename [input filename]
```

strip470	is the command that invokes the strip utility.
<i>input filename</i>	is a COFF object file (.obj) or an executable file (.out).
<i>options</i>	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 name utility option is as follows: <ul style="list-style-type: none">-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 TMS470R1x 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 more 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

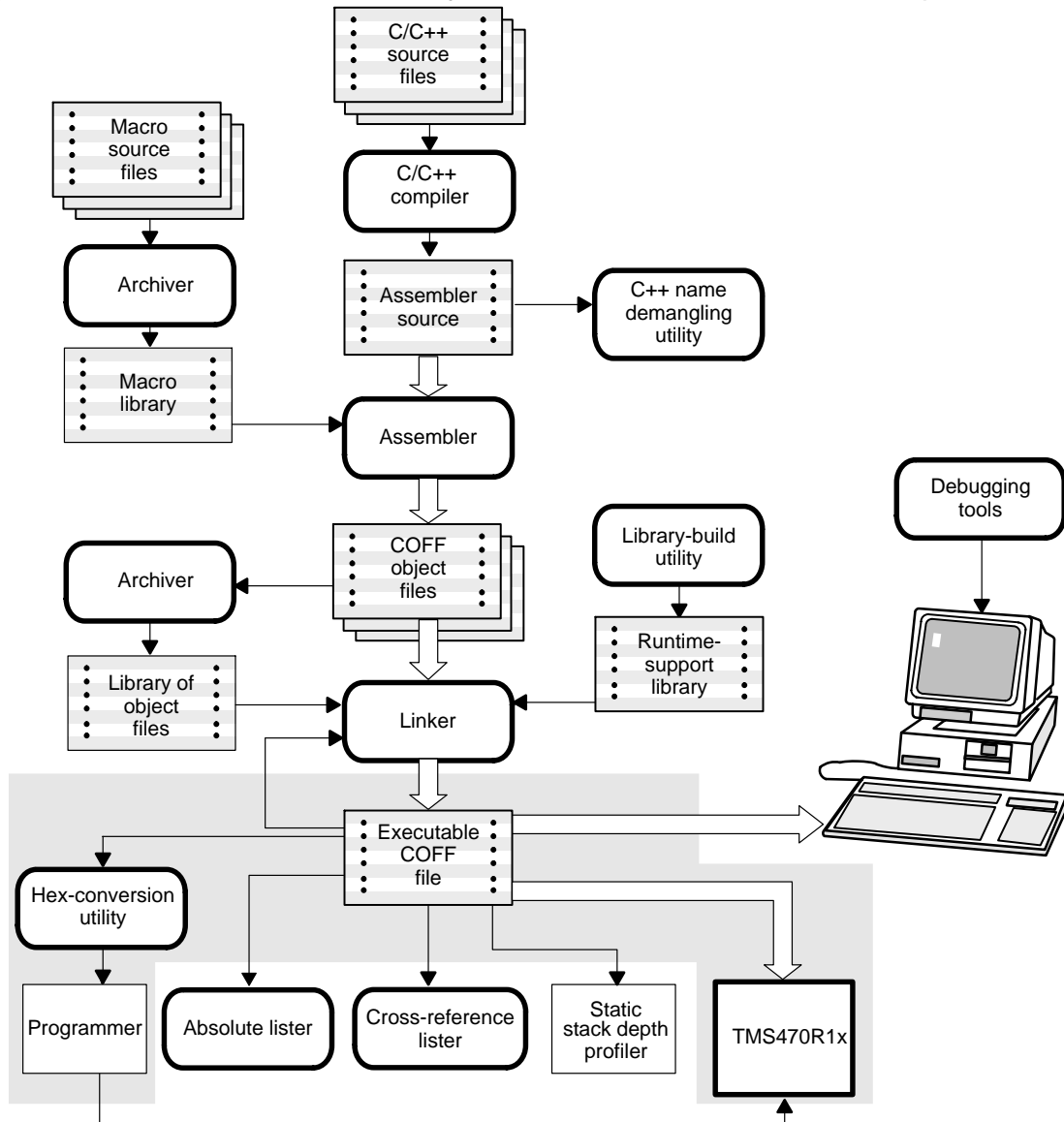
This chapter explains how to invoke the hex utility to convert a COFF object file into one of several standard hexadecimal formats suitable for loading into an EPROM programmer.

Topic	Page
11.1 The Hex Conversion Utility's Role in the Software Development Flow	11-2
11.2 Invoking the Hex Conversion Utility	11-3
11.3 Understanding Memory Widths	11-8
11.4 The ROMS Directive	11-13
11.5 The SECTIONS Directive	11-19
11.6 Assigning Output Filenames	11-20
11.7 Image Mode and the -fill Option	11-22
11.8 Controlling the ROM Device Address	11-24
11.9 Description of the Object Formats	11-25

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 TMS470R1x 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`.

```
hex470 -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`:

```
hex470 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:

hex470 *[options] filename*

hex470 is the command that invokes the hex conversion utility.

options supplies additional information that controls the hex conversion process. You can use options on the command line or in a command file; see Table 11–1 for a list of options. When using options the following items apply:

- ❑ All options are preceded by a dash and are not case-sensitive.
- ❑ Several options have an additional parameter that must be separated from the option by at least one space.
- ❑ Options with multicharacter names must be spelled exactly as shown in this document; no abbreviations are allowed.
- ❑ Options are not affected by the order in which they are used. The exception to this rule is the `-q` option, which must be used before any other options.

filename names a COFF object file or a command file (for more information, see section 11.2.2, *Invoking the Hex Conversion Utility With a Command File*, on page 11-6). If you do not specify a filename, the hex conversion utility prompts for one.

Table 11–1. Hex Conversion Utility Options

General Options	Option	Description	Page
Control the overall operation of the hex conversion utility.	–boot	Select boot mode	
	–byte	Number output locations by bytes rather than by target addressing	
	–datawidth <i>number</i>	Represents the width, in bits, of data words	
	–map <i>filename</i>	Generate a map file	11-17
	–o <i>filename</i>	Specify an output filename	11-21
	–olength <i>number</i>	Set the number of data items per line of hex output	
	–order LS	Indicate data ordering to be little endian	
	–order MS	Indicate data ordering to be big endian	
	–q	Run quietly (when used, it must appear <i>before</i> other options)	11-6
Image Options	Option	Description	Page
Create a continuous image of a range of target memory.	–fill <i>value</i>	Fill holes with <i>value</i>	11-23
	–image	Specify image mode	11-22
	–zero	Reset the address origin to zero in image mode	11-24
Memory Options	Option	Description	Page
Configure the memory widths for your output files.	–memwidth <i>value</i>	Define the system memory word width (default 32 bits)	11-9
	–romwidth <i>value</i>	Specify the ROM device width (default depends on output format used)	11-10
Output Formats	Option	Description	Page
Specify the output format.	–a	Select ASCII-Hex	11-26
	–i	Select Intel	11-27
	–m	Select Motorola-S	11-28

Table 11–1. Hex Conversion Utility Options (Continued)

Output Formats	Option	Description	Page
	–m1	Indicates a preference for S1 records	11-28
	–m2	Indicates a preference for S2 records	11-28
	–m3	Indicates a preference for S3 records	11-28
Specify the output format.	–t	Select TI-Tagged	11-29
	–x	Select Tektronix (default)	11-30

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.

Hex conversion utility 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, *The ROMS Directive*, on page 11-13.)
- ☐ **SECTIONS directive.** The hex conversion utility SECTIONS directive specifies which sections from the COFF object file should be selected. (For more information, see section 11.5, *The Sections Directive*, on page 11-19.)
- ☐ **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:

hex470 *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:

```
hex470 firmware.cmd -map firmware.mxp
```

The order in which these options and file names 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 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:

```
hex470 firmware.cmd
```

The following 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

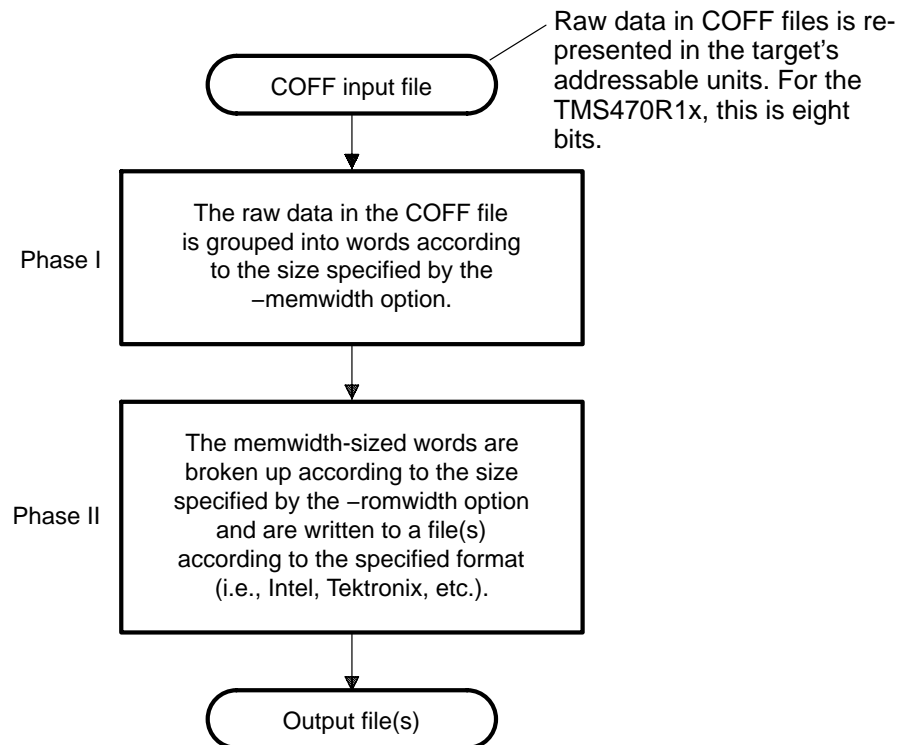
The hex conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex conversion utility, *you must understand how the utility treats word widths*. Three widths are important in the conversion process:

- ☐ 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 TMS470R1x 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, narrower memory words.

The hex conversion utility defaults memory width to the target width (in this case, 32 bits).

You can change the memory width by:

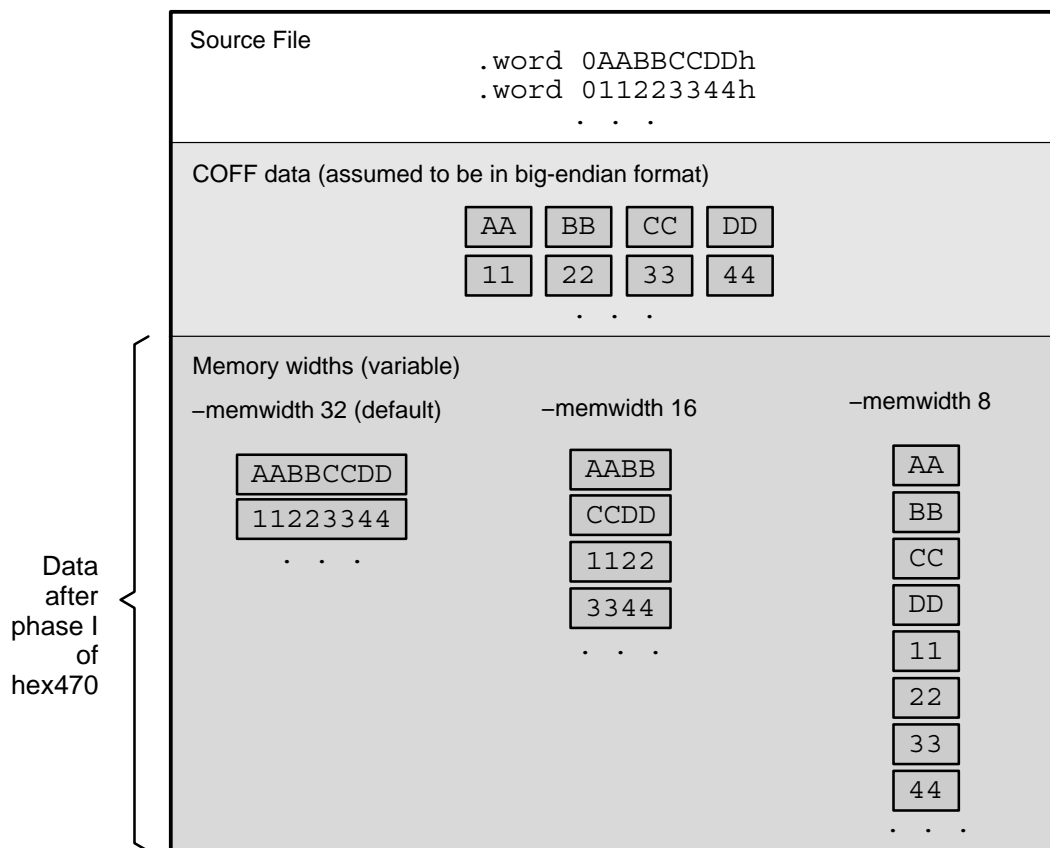
- ☐ Using the **-memwidth** option. This changes the memory width value for the entire file.
- ☐ Setting the **memwidth** parameter of the ROMS directive. This changes the memory width value for the address range specified in the ROMS directive and overrides the **-memwidth** option for that range. See section 11.4, *The ROMS Directive*, on page 11-13.

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.

Note: The TI-Tagged Format Is 16 Bits Wide

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, *The ROMS Directive*, on page 11-13.

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. When you refer to bits of memory, notice that the bits of the memory word are always numbered from right to left as follows:

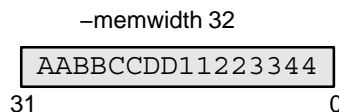
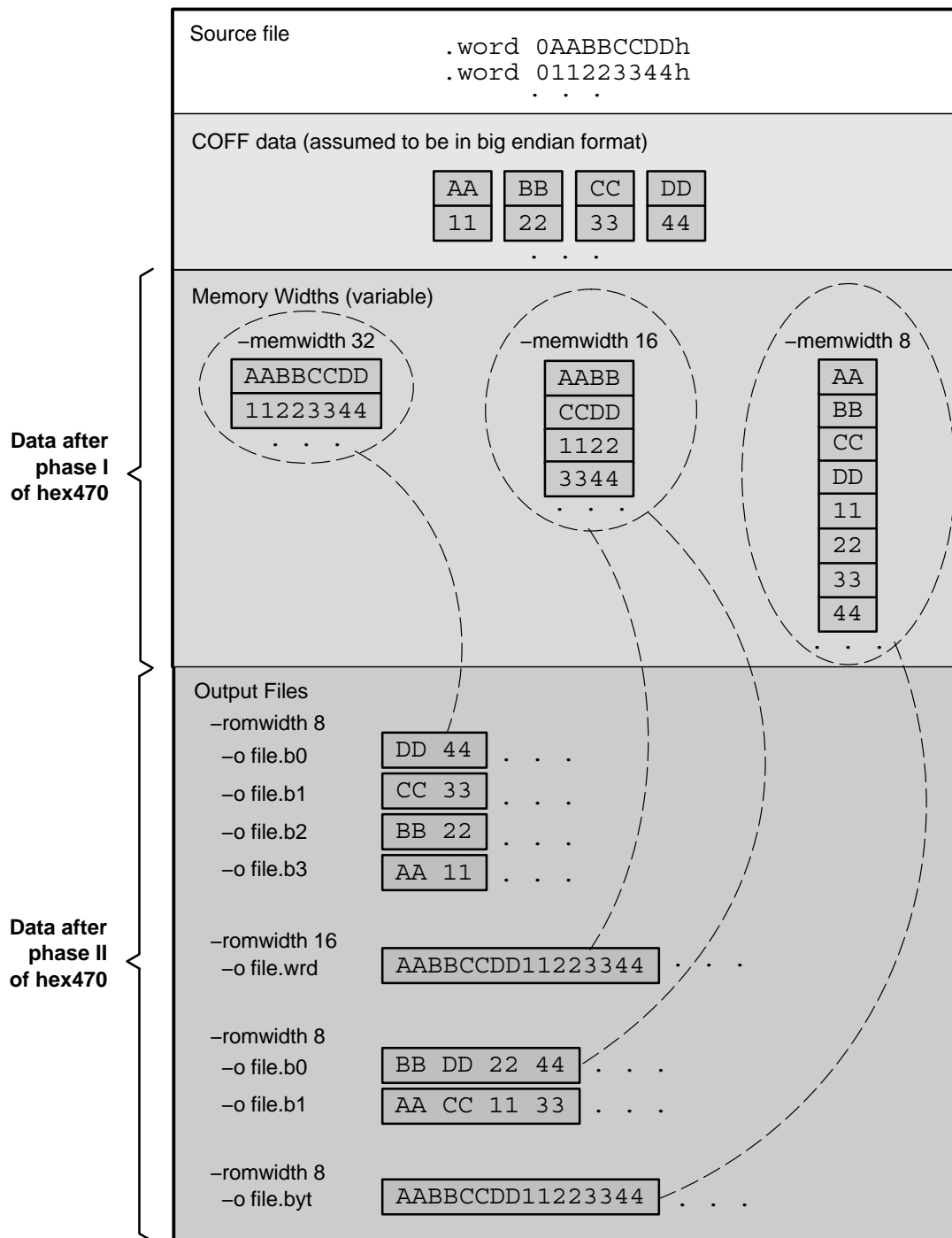


Figure 11–4. Data, Memory, and ROM Widths



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 TMS470R1x 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, ...}]

    ...
}
```

ROMS begins the directive definition.

romname identifies a memory range. The name of the memory range may 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).

origin 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

length	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.
romwidth	specifies the physical ROM width of the range in bits (see section 11.3.3, <i>Partitioning Data Into Output Files</i> , on page 11-10). Any value you specify here overrides the <code>-romwidth</code> option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8.
memwidth	specifies the memory width of the range in bits (see section 11.3.2, <i>Specifying the Memory Width</i> , on page 11-9). Any value you specify here overrides the <code>-memwidth</code> 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</i> (see page 11-19).
fill	<p>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.</p> <p>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 <code>-fill</code> option. When using fill, you must also use the <code>-image</code> command line option. See section 11.7.2, <i>Specifying a Fill Value</i>, on page 11-23.</p>
files	<p>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.</p> <p>The number of file names should equal the number of output files that the range will generate. To calculate the number of output files, refer to section 11.3.3, <i>Partitioning Data Into Output Files</i>, on page 11-10. The utility warns you if you list too many or too few filenames.</p>

Unless you are using the `-image` option (see section 11.7 on page 11-22), 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. In this way, 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 $8K \times 8$ -bit EPROMs. Figure 11–5 illustrates the input and output files.

Example 11–1. A ROMS Directive Example

```

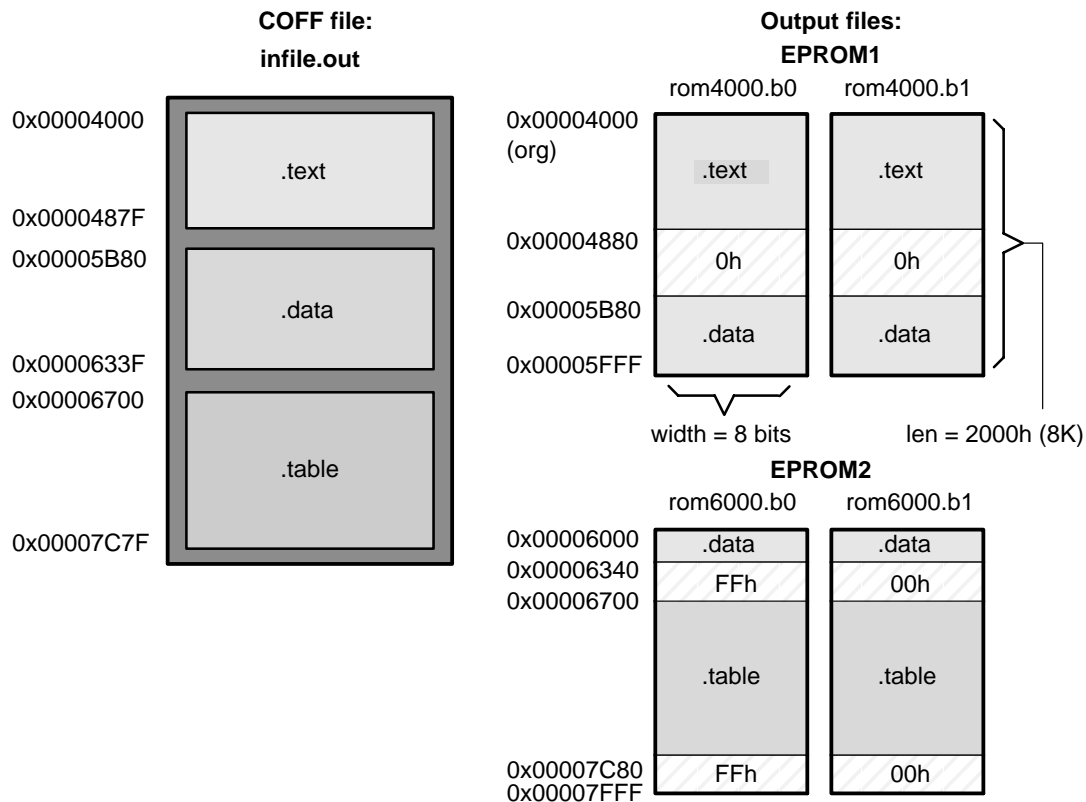
infile.out
-image
-memwidth 16

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 Example 11–1.

Example 11–2. Map File Output From Example 11–1 That Shows 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

```

EPROM1 defines the address range from 0x00004000 through 0x00005FFF. The range contains 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). The data from this range is 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. The range contains 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 four 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. The TMS470R1x compiler-generated initialized sections are: .text, .const, and .cinit.

Uninitialized sections are *never* converted, whether or not you specify them in a SECTIONS directive.

Note: Sections Generated by the C Compiler

The TMS470R1x C compiler automatically generates these sections:

- ☐ **Initialized sections:** .text, .const, and .cinit.
- ☐ **Uninitialized sections:** .bss, .stack, and .system.

Use the SECTIONS directive in a command file. (For more information, see section 11.2.2, *Invoking the Hex Conversion Utility With a Command File*, on page 11-6.) The general syntax for the SECTIONS directive is:

```
SECTIONS
{
    sname[:] [paddr=value][,]
    sname[:] [paddr=value][,]
    ...
}
```

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.

paddr=value specifies the physical ROM address at which this section should be located. This value overrides the section load address given by the linker. The value must be a decimal, octal, or hexadecimal constant. If one section uses this option, then all sections must use the option.

For more similarity with the linker's SECTIONS directive, you can use colons after the section names. For example, the data in your application (section partB) must be loaded on the EPROM at address 0x0. Use the paddr option with the SECTIONS directive to specify this:

```
SECTIONS
{
    partB:  paddr = 0x0
}
```

The commas separating section names are optional. For example, 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 }
```

11.6 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 (LSBs) to xyz.b0 and the most significant bits (MSBs) 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 11.9 on page 11-25 for format descriptions):

a for ASCII-Hex
i for Intel
m for Motorola-S
t for TI-Tagged
x 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 the filename 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 this data ...
<code>coff.i0o</code> , <code>coff.i01</code> , <code>coff.i02</code> , and <code>coff.i03</code>	0x00001000 through 0x00001FFF
<code>coff.i10</code> , <code>coff.i11</code> , <code>coff.i12</code> , and <code>coff.i13</code>	0x00002000 through 0x00002FFF

11.7 Image Mode and the *-fill* Option

This section highlights the advantages of operating in image mode and describes how to produce output files with a precise, continuous image of a target memory range.

11.7.1 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. Hence, 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.

Note: Defining the Ranges of Target Memory

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.

11.7.2 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.*

11.7.3 Steps to Follow in Using Image Mode

- Step 1:** Define the ranges of target memory with a ROMS directive. See section 11.4, *The ROMS Directive*, on page 11-13 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 zero 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 zero, use the `-fill` option.

11.8 Controlling the ROM Device Address

The hex conversion utility output address corresponds to the ROM device address. The EPROM programmer burns the data in the location specified by the address field in the hex conversion utility output file. 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 where the data will be burned.

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 a hex conversion utility output file is the load address (as given in the linker command file).
- 2) **The `paddr` option inside the `SECTIONS` directive.** When the `paddr` option is specified for a section (described on page 11-19), 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 zero for each output file. Since each file starts at zero and counts upward, any address record represents offsets from the beginning of the file (the address within 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 option.

11.9 Description of the Object Formats

The hex conversion utility has options that identify each format and Table 11–2 lists these options. They are detailed in the following sections.

- ☐ 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 11–2. 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

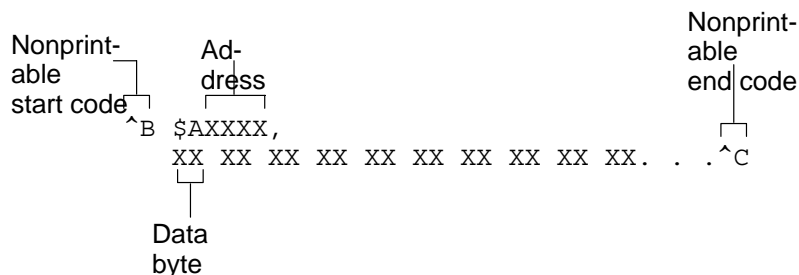
Address bits determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

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.

11.9.1 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 \$Axxxx, in which xxxx is a 4-digit (16-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.

11.9.2 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 prefix—four fields that define 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

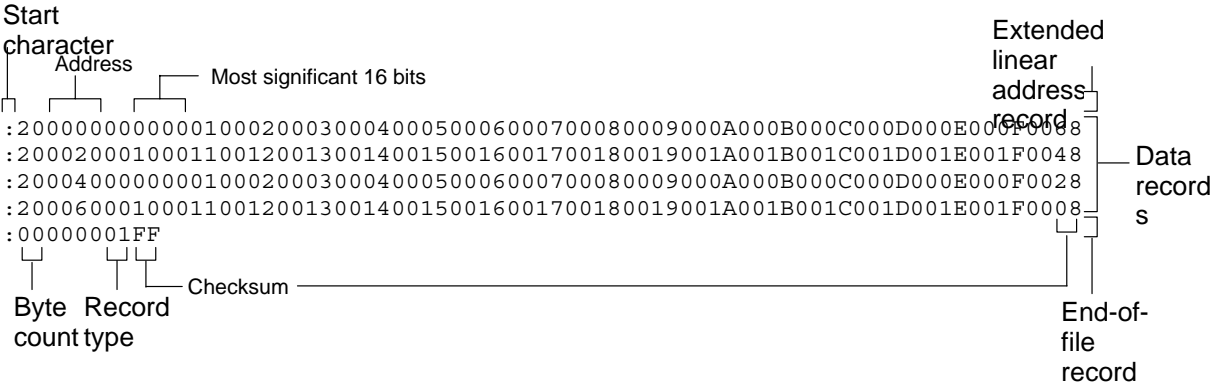
Record type 00, the data record, begins with a colon (:) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended 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.

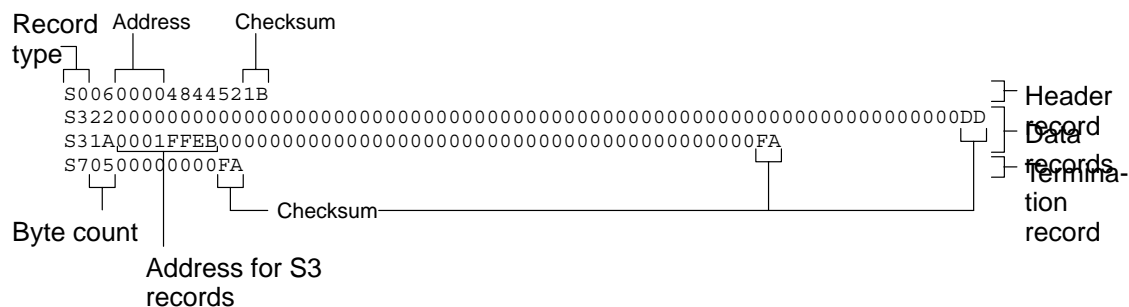
Record type 04, the extended linear address record, specifies the upper 16 address bits. It begins with a colon, followed by the byte count, a dummy address of 0h, the record type (04), the most significant 16 bits of the address, and the checksum. The subsequent address fields in the data records contain the least significant bits of the address.

Figure 11–7 illustrates the Intel hexadecimal object format.

Figure 11–7. Intel Hexadecimal Object Format



Record Type	Description
S0	Header record
S3	Code/data record
S7	Termination record



11.9.5 Extended Tektronix Object Format (-x Option)

The Tektronix object format supports 32-bit addresses and has two types of records:

Data record contains the header field, the load address, and the object code.

Termination record signifies the end of a module.

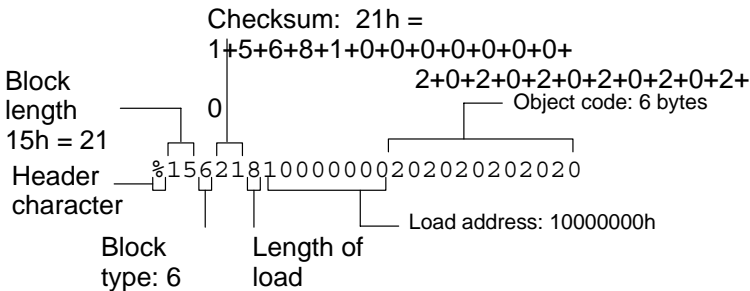
The header field in the data record contains the following information:

Item	Number of ASCII Characters	Description
%	1	Data type is Tektronix format
Block length	2	Number of characters in the record, minus the %
Block type	1	6 = data record 8 = termination record
Checksum	2	A 2-digit hex sum modulo 256 of all values in the record except the % and the checksum itself

The load address in the data record specifies where the object code will be located. The first digit specifies the address length; this is always 8. The remaining characters of the data record contain the object code, two characters per byte.

Figure 11–10 illustrates the Tektronix object format.

Figure 11–10. Extended Tektronix Object Format



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 is used because it 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, *Introduction to Common Object File Format*, 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 TMS470R1x 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
A.1 COFF File Structure	A-2
A.2 File Header Structure	A-4
A.3 Optional File Header Format	A-6
A.4 Section Header Structure	A-7
A.5 Structuring Relocation Information	A-10
A.6 Symbol Table Structure and Content	A-12

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
- ☐ Line-number entries for each initialized section
- ☐ A symbol table
- ☐ A string table

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

Figure A–1. COFF File Structure

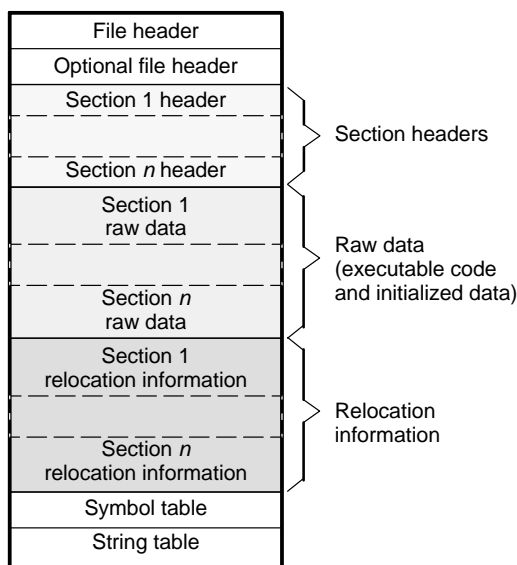
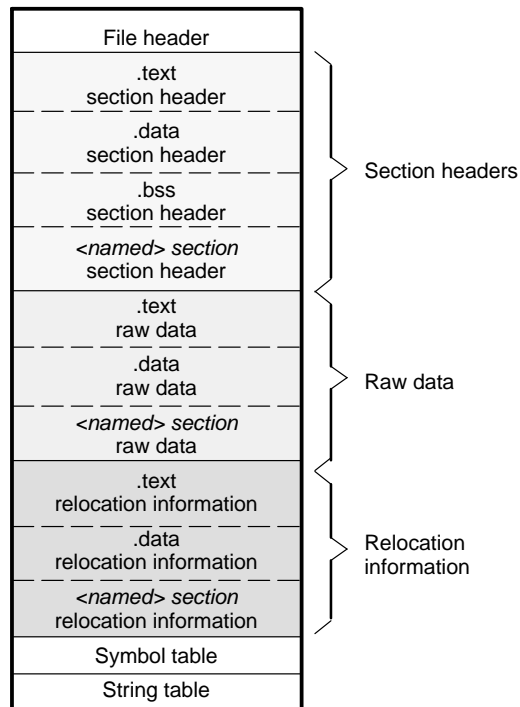


Figure A–2 shows a typical example of a COFF object file that contains the three default sections, .text, .data, and .bss, and a named section (referred to as <named>). By default, the tools place sections into the object file in the following order: .text, .data, initialized named sections, .bss, and 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 20 bytes of information that describe the general format of an object file. Table A–1 shows the structure of the Version 2 COFF file header.

Table A–1. File Header Contents

Byte Number	Type	Description
0–1	Unsigned short integer	Version ID; indicates version of COFF file structure
2–3	Unsigned short integer	Number of section headers
4–7	Long integer	Time and date stamp; indicates when the file was created
8–11	Long integer	File pointer; contains the symbol table's starting address
12–15	Long integer	Number of entries in the symbol table
16–17	Unsigned short integer	Number of bytes in the optional header. This field is either 0 or 28; if it is 0, there is no optional file header.
18–19	Unsigned short integer	Flags (see Table A–2)
20–21	Unsigned short integer	Target ID; magic number (0097h) indicates the file can be executed in a TMS470R1x system

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

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

Mnemonic	Flag	Description
F_RELFLG	0001h	Relocation information was stripped from the file.
F_EXEC	0002h	The file is relocatable (it contains no unresolved external references).
F_LNNO	0004h	Line numbers were stripped from the file.
F_LSYMS	0008h	Local symbols were stripped from the file.
F_LITTLE	0100h	The file has little-endian byte ordering (least significant byte first).
F_BIG	0200h	The file has big-endian byte ordering (most significant byte first).
F_SYMMERGE	1000h	Duplicate symbols were removed.

A.3 Optional File Header Format

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

Table A–3. Optional File Header Contents

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

A.4 Section Header Structure

COFF object files contain a table of section headers that 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	Long integer	Section's physical address
12–15	Long integer	Section's virtual address
16–19	Long integer	Section size in bytes
20–23	Long integer	File pointer to raw data
24–27	Long integer	File pointer to relocation entries
28–31	Long integer	File pointer to line number entries
32–35	Unsigned long integer	Number of relocation entries
36–39	Unsigned long integer	Number of line number entries
40–43	Unsigned long integer	Flags (see Table A–5)
44–45	Unsigned short integer	Reserved
46–47	Unsigned short integer	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 36 Through 39)

Mnemonic	Flag	Description
STYP_REG	0000 0000h	Regular section (allocated, relocated, loaded)
STYP_DSECT	0000 0001h	Dummy section (relocated, not allocated, not loaded)
STYP_NOLOAD	0000 0002h	Noload section (allocated, relocated, not loaded)
STYP_XXX1	0000 0004h	Reserved
STYP_XXX2	0000 0008h	Reserved
STYP_COPY	0000 0010h	Copy section (relocated, loaded, but not allocated; relocation and line number entries are processed normally)
STYP_TEXT	0000 0020h	Section contains executable code
STYP_DATA	0000 0040h	Section contains initialized data
STYP_BSS	0000 0080h	Section contains uninitialized data
STYP_BLOCK	0000 1000h	Alignment used as a blocking factor
STYP_PASS	0000 2000h	Section should pass through unchanged
STYP_CLINK	0000 4000h	Section requires conditional linking

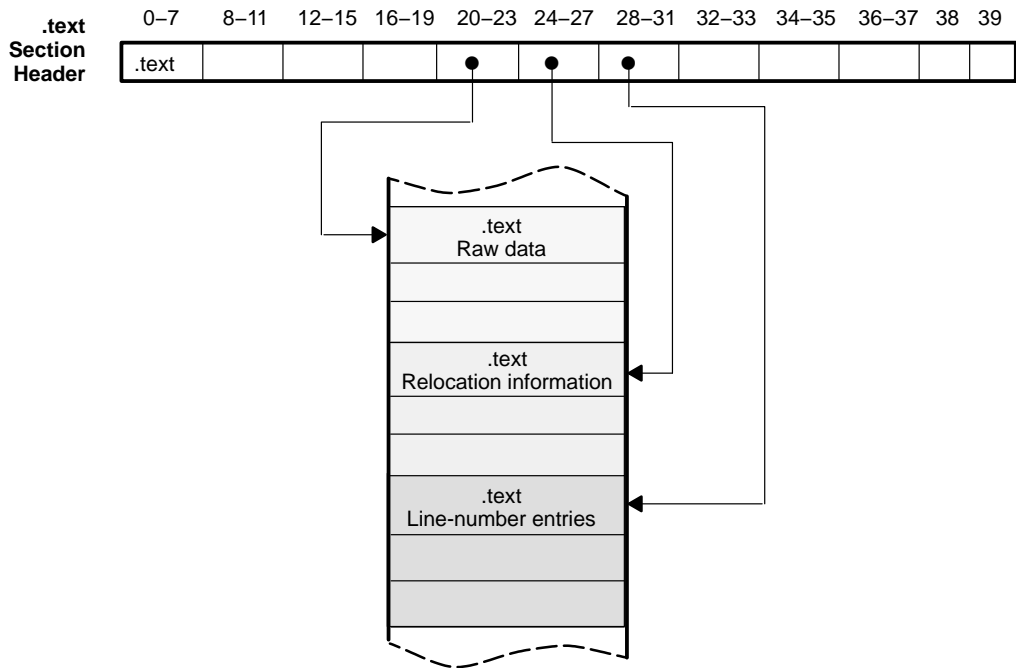
Notes:

- 1) The term *loaded* means that the raw data for this section appears in the object file.
- 2) Alignment is indicated by the bits masked by 0xF00. Alignment is the value in the bits raised to a power equal to the bit value. Alignment is 2 raised to the same power. For example, if the value in these 4 bits is 2, the alignment is 2 raised to the power 2 (or 4).

The flags listed in Table A-5 can be combined; for example, if the flag's word is set to 024h, both STYP_GROUP and STYP_TEXT are set.

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.

Figure A-3. Section Header Pointers for the .text Section



As Figure A-2 on page A-3 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, relocation information, or line number information. They occupy no actual space in the object file. Therefore, the number of relocation entries, the number of line number entries, and the file pointers are 0 for an uninitialized section. The header of an uninitialized section simply tells the linker how much space for variables it should reserve in the memory map.

A.5 Structuring Relocation Information

A COFF object file has one relocation entry for each relocatable reference. The assembler automatically generates relocation entries. The linker reads the relocation entries as it reads each input section and performs relocation. The relocation entries determine how references within 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	Long integer	Virtual address of the reference
4-5	Unsigned short integer	Symbol table index (0-65 535)
6-7	Unsigned short integer	Reserved
8-9	Unsigned short integer	Relocation type (see Table A-7)

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 EAFFFFFFE!          B          x

```

In this example, the virtual address of the relocatable field is 0.

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

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

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

The **relocation type** specifies the size of the field to be patched and describes how the patched value should be calculated. The type field depends on the addressing mode that was used to generate the relocatable reference. In the preceding example, the actual address of the referenced symbol *x* will be placed in a 23-bit field in the object code. This is a 23-bit PC-relative relocation, so the relocation type is R_PCR23H. Table A–7 lists the relocation types.

Table A–7. Relocation Types (Bytes 8 and 9)

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

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
⋮
Defined global symbols
Undefined global symbols

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

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

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

Section names are also defined in the symbol table.

All symbol entries, regardless of class and type, have the same format in the symbol table. Each symbol table entry contains the 18 bytes of information listed in Table A-8. Each symbol may also have an 18-byte auxiliary entry; the special symbols listed in Table A-9 on page A-13 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	Character	This field contains one of the following: <ol style="list-style-type: none"> 1) An 8-character symbol name, padded with nulls 2) A pointer into the string table if the symbol name is longer than eight characters
8–11	Long integer	Symbol value; storage class dependent
12–13	Short integer	Section number of the symbol
14–15	Unsigned short integer	Basic and derived type specification
16	Character	Storage class of the symbol
17	Character	Number of auxiliary entries (always 0 or 1)

A.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 long integers. The entire symbol name is stored in the string table. Bytes 0–3 contain 0, and bytes 4–7 are an offset into the string table.

A.6.3 String Table Structure

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

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

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 the section in which the symbol was defined. Table A–11 lists these numbers and the sections they indicate.

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 TMS470R1x 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.

Topic	Page
B.1 DWARF Debugging Format	B-2
B.2 COFF Debugging Format	B-3
B.3 Debug Directive Syntax	B-4

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:

```
cl470 -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:

```
cl470 --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 *TMS470R1x Optimizing C/C++ Compiler User's Guide*.

Table B–1. Symbolic Debugging Directives

Label	Directive	Arguments
	.block	[beginning line number]
	.dwattr	DIE label, DIE attribute name(DIE attribute value)[,DIE attribute name(attribute value) [, ...]
	.dwcfa	call frame instruction opcode[, operand[, operand]]
CIE label	.dwcie	version, return address register
	.dwendentry	
	.dwendtag	
	.dwfde	CIE label
	.dwpsn	“filename”, line number, column number
DIE label	.dwtag	DIE tag name, DIE attribute name(DIE attribute value)[,DIE attribute name(attribute value) [, ...]
	.endblock	[ending line number]
	.endfunc	[ending line number[, register mask[, frame size]]]
	.eos	
	.etag	name[, size]
	.file	“filename”
	.func	[beginning line number]
	.line	line number[, address]
	.member	name, value[, type, storage class, size, tag, dims]
	.stag	name[, size]
	.sym	name, value[, type, storage class, size, tag, dims]
	.utag	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
C.1 XML Information File Element Types	C-2
C.2 Document Elements	C-3

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>** element lists the time of the link execution (string).
- ☐ The **<link_timestamp>** 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>TMS470 COFF Linker          Version x.xx (Jan  6 2003)</banner>
<copyright>Copyright (c) 1996-2003 Texas Instruments Incorporated</copyright>
<link_time>Mon Jan  6 15:38:18 2003</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="fl-1">
    <kind>object</kind>
    <file>hi.obj</file>
    <name>hi.obj</name>
  </input_file>
  <input_file id="fl-2">
    <path>/tools/lib/</path>
    <kind>archive</kind>
    <file>rtsxxx.lib</file>
    <name>boot.obj</name>
  </input_file>
  <input_file id="fl-3">
    <path>/tools/lib/</path>
    <kind>archive</kind>
    <file>rtsxxx.lib</file>
    <name>exit.obj</name>
  </input_file>
  <input_file id="fl-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 overlay (reference).
 - The **<logical_group_ref>** is a logical group that is contained in this overlay (reference).
- The **<split_section>** is another special kind of logical group which 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>** element lists elements contained in this split section (container). 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):
 - The **<start_address>** specifies the address of the fragment (constant).
 - The **<size>** specifies the size of the fragment (constant).
 - The **<logical_group_ref>** provides a reference to the logical group that is allocated to this fragment (reference).
 - The **<available_space>** element provides details of an available fragment within this memory area (container):
 - The **<start_address>** specifies the address of the fragment (constant).
 - The **<size>** specifies the size of the fragment (constant).

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 TMS470R1x 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>
```

Hex Conversion Utility Examples

The flexible hex conversion utility offers many options and capabilities. Once you understand the proper ways to configure your EPROM system and the requirements of the EPROM programmer, you will find that converting a file for a specific application is easy.

The three scenarios in this appendix show how to develop a hex conversion command file for avoiding holes, using 16-BIS (16-bit instruction set) code, and using multiple-EPROM systems. The scenarios use the assembly code shown in Example D–1.

Example D–1. Assembly Code for Hex Conversion Utility Scenarios

```
*****
* Assemble two words into section "secA"          *
*****

.sect "secA"
.word 012345678h
.word 0abcd1234h

*****
* Assemble two words into section "secB"          *
*****

.sect "secB"
.word 087654321h
.word 04321dcba
```

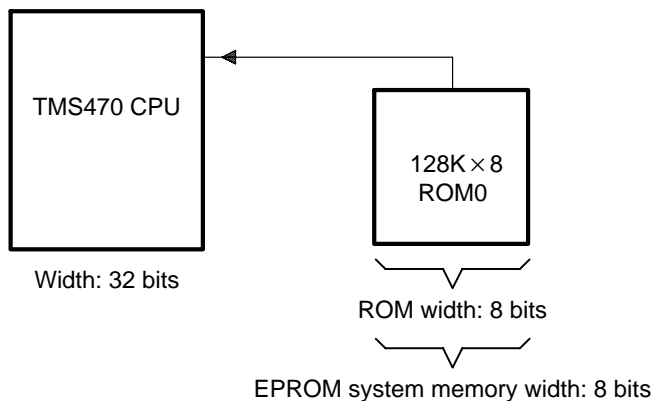
Before you use this appendix, read Chapter 11, *Hex Conversion Utility Description*, to understand how to use the hex conversion utility.

Topic	Page
D.1 Scenario 1: Building a Hex Conversion Command File for a Single 8-Bit EPROM	D-2
D.2 Scenario 2: Building a Hex Conversion Command File for 16-BIS Code	D-8
D.3 Scenario 3: Building a Hex Conversion Command File for Two 8-Bit EPROMs	D-12

D.1 Scenario 1: Building a Hex Conversion Command File for a Single 8-Bit EPROM

Scenario 1 shows how to build the hex conversion command file for converting a COFF object file for the memory system shown in Figure D–1. In this system, there is one external 128K × 8-bit EPROM interfacing with a TMS470 target processor.

Figure D–1. EPROM Memory System for Scenario 1



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. Scenario 1 shows how you can use the hex conversion utility's image mode to fill any holes before, between, or after sections with a fill value.

For this scenario, the application code resides in the program memory (ROM) on the TMS470 CPU, but the data tables used by this code reside in an off-chip EPROM.

The circuitry of the target board handles the access to the data; the native TMS470 address of 0x1000 accesses location 0x0 on the EPROM.

To satisfy the address requirements for the code, this scenario requires a linker command file that allocates sections and memory as follows:

- ❑ The program/application code (represented in this scenario by the `secA` section shown in Example D–1) must be linked so that its address space resides in the program memory (ROM) on the TMS470 CPU.

- ❑ To satisfy the condition that the data be loaded on the EPROM at address 0x0 but be referenced by the application code at address 0x1000, secB (the section that contains the data for this application) must be assigned a linker load address of 0x1000 so that all references to data in this section will be resolved with respect to the TMS470 CPU address. In the hex conversion utility command file, the paddr option must be used to burn the section of data at EPROM address 0x0. This value overrides the section load address given by the linker.

Example D–2 shows the linker command file that resolves the addresses needed in the stated specifications.

Example D–2. Linker Command File and Link Map for Scenario 1

```

/*****
/* Scenario 1 Link Command
/*
/* Usage: lnk470 <obj files...> -o <out file> -m <map file> lnk32.cmd
/*         cl470 <src files...> -z -o <out file> -m <map file> lnk32.cmd
/*
/* Description: This file is a sample command file that can be used
/*              for linking programs built with the TMS470 C
/*              compiler. Use it as a guideline; you may want to change
/*              the allocation scheme according to the size of your
/*              program and the memory layout of your target system.
/*
/* Notes: (1) You must specify the directory in which rts32.lib is
/*           located. Either add a "-i<directory>" line to this
/*           file, or use the system environment variable C_DIR to
/*           specify a search path for libraries.
/*
/*           (2) If the runtime-support library you are using is not
/*           named rts32.lib, be sure to use the correct name here.
*****/

-m example1.map

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
    I_MEM    : org = 0x00000000    len = 0x00000020 /* INTERRUPTS          */
    D_MEM    : org = 0x00000020    len = 0x00010000 /* DATA MEMORY (RAM) */
    P_MEM    : org = 0x00010020    len = 0x00100000 /* PROGRAM MEMORY (ROM)
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    secA: load = P_MEM
    secB: load = 0x1000
}

```

You must create a hex conversion command file to generate a hex output with the correct addresses and format for the EPROM programmer.

In the memory system outlined in Figure D–1, only the application data is stored on the EPROM; the data resides in secB of the object file created by the linker. By default, the hex conversion utility converts all initialized sections that appear in the COFF object file. To prevent the conversion of the application code in secA, a SECTIONS directive must be defined in the hex conversion command file to list explicitly the section(s) to be converted. In this case, secB must be listed explicitly as the section to be converted.

The EPROM programmer in this scenario has the following system requirements:

- ☐ The EPROM programmer loads only a complete ROM image. A complete ROM image is one in which there is a contiguous address space (there are no holes in the addresses in the converted file), and each address in the range contains a known value. Creating a complete ROM image requires the use of the `-image` option and the `ROMS` directive.
 - Using the `-image` option causes the hex conversion utility to create an output file that has contiguous addresses over the specified memory range and forces the utility to fill address spaces that are not previously filled by raw data from sections defined in the input COFF object file. By default, the value used to fill the unused portions of the memory range is 0.
 - Because the `-image` option operates over a known range of memory addresses, a `ROMS` directive is needed to specify the origin and length of the memory for the EPROM.
- ☐ To burn the section of data at EPROM address 0x0, the `paddr` option must be used. This value overrides the section load address given by the linker.
- ☐ In this scenario, the EPROM is 128K×8 bits. Therefore, the memory addresses for the EPROM must range from 0x0 to 0x20000.
- ☐ Because the EPROM memory width is eight bits, the `memwidth` value must be set to 8.
- ☐ Because the physical width of the ROM device is eight bits, the `romwidth` value must be set to 8.
- ☐ Intel format must be used.

Since `memwidth` and `romwidth` have the same value, only one output file is generated (the number of output files is determined by the ratio of `memwidth` to `romwidth`). The output file is named with the `-o` option.

The hex conversion command file for Scenario 1 is shown in Example D–3. This command file uses the following options to select the requirements of the system:

Option	Description
-i	Create Intel format
-image	Generate a memory image
-map example1.mxp	Generate example1.mxp as the map file of the conversion
-o example1.hex	Name example1.hex as the output file
-memwidth 8	Set EPROM system memory width to 8
-romwidth 8	Set physical ROM width to 8

Example D–3. Hex Conversion Command File for Scenario 1

```
/* Hex Conversion Command file for Scenario 1          */
a.out          /* linked COFF object file, input */
-i            /* Intel format */
-image
-map example1.mxp /* Generate a map of the conversion */
-o example1.hex  /* Resulting hex output file */

-memwidth 8    /* EPROM memory system width */
-romwidth 8    /* Physical width of ROM */

ROMS
{
    EPROM: origin = 0x0, length = 0x20000
}

SECTIONS
{
    secB: paddr = 0x0 /* Select only section, secB, for conversion */
}
```

Example D-4 shows the contents of the resulting map file (example1.mxp). Example D-5 shows the contents of the resulting hex output file (example1.hex). The hex conversion utility places the data tables, secB, at address 0 and then fills the remainder of the address space with the default fill value of 0. For more information about the Intel MCS-86 object format, see section 11.9.2 on page 11-27.

Example D-4. Contents of Hex Map File example1.mxp

```
*****
TMS470 COFF/Hex Converter          Version x.xx
*****
Mon Sep 18 15:57:00 1995

INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
  Default data width:      8
  Default memory width:    8
  Default output width:    8

OUTPUT TRANSLATION MAP
-----
00000000..0001ffff  Page=0  ROM Width=8  Memory Width=8  "EPROM"
-----
  OUTPUT FILES: example1.hex [b0..b7]

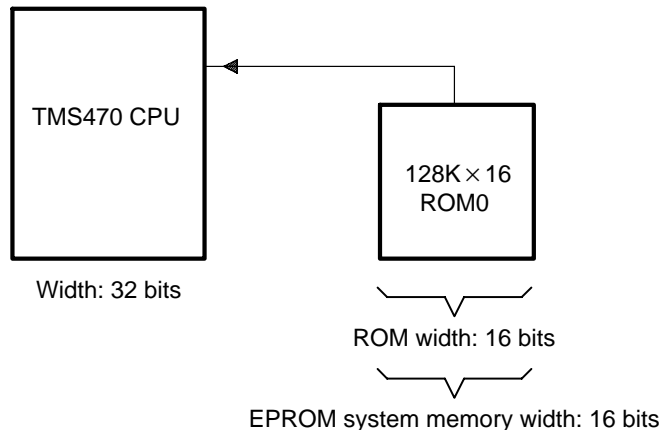
  CONTENTS: 00000000..00000007  Data Width=1  secB
            00000007..0001ffff  FILL = 00000000
```

The diagram illustrates the SEC-B file format structure. It begins with a 'Start' marker, followed by a 'character' field. The main body consists of a series of records, each containing an 'Address' field and 'secB data tables'. The records are separated by a 'Record type' field. The structure ends with an 'End-of-file record' and a 'Checksum' field. The diagram shows a sequence of records with addresses ranging from 20000000 to 20FFC000, with a final record at 20FFC000. The 'secB data tables' field is shown as a large block of data. The 'Record type' field is shown as a small block of data. The 'End-of-file record' is shown as a record with address 20FFC000 and a 'Record type' of 01FF. The 'Checksum' field is shown as a final field in the file.

D.2 Scenario 2: Building a Hex Conversion Command File for 16-BIS Code

Scenario 2 shows how to build the hex conversion command file to generate the correct converted file for the application code and data that will reside on a single 16-bit EPROM. The EPROM memory system for this scenario is shown in Figure D–2. For this scenario, the TMS470 CPU operates with the T control bit set, so the processor executes instructions in 16-BIS mode.

Figure D–2. EPROM Memory System for Scenario 2



For this scenario, the application code and data reside on the EPROM: the lower 64K words of EPROM memory are dedicated to application code space and the upper 64K words are dedicated to the data tables. The application code is loaded starting at address 0x0 on the EPROM but maps to the TMS470 CPU at address 0x3000. The data tables are loaded starting at address 0x1000 on the EPROM and map to the TMS470 CPU address 0x20.

Example D–6 shows the linker command file that resolves the addresses needed for the load on EPROM and the TMS470 CPU access.

Example D–6. Linker Command File for Scenario 2

```
/* ***** */
/* Scenario 2 Link Command */
/*
/* Usage: lnk470 <obj files...> -o <out file> -m <map file> lnk16.cmd */
/*         cl470 <src files...> -z -o <out file> -m <map file> lnk16.cmd */
/*
/* Description: This file is a sample command file that can be used */
/*               for linking programs built with the TMS470 C */
/*               compiler. Use it as a guideline; you may want to change */
/*               the allocation scheme according to the size of your */
/*               program and the memory layout of your target system. */
/*
/* Notes: (1) You must specify the directory in which rts16.lib is */
/*            located. Either add a "-i<directory>" line to this */
/*            file, or use the system environment variable C_DIR to */
/*            specify a search path for libraries. */
/*
/*            (2) If the runtime-support library you are using is not */
/*            named rts16.lib, be sure to use the correct name here. */
/* ***** */

-m example2.map

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
    I_MEM : org = 0x00000000 len = 0x00000020 /* INTERRUPTS */
    D_MEM : org = 0x00000020 len = 0x00010000 /* DATA MEMORY (RAM) */
    P_MEM : org = 0x00010020 len = 0x00100000 /* PROGRAM MEMORY (ROM) */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    secA: load = 0x3000
    secB: load = 0x20
}
```

You must create a hex conversion command file to generate a hex output with the correct addresses and format for the EPROM programmer. The EPROM programmer in this scenario has the following system requirements:

- ☐ Because the EPROM memory width is 16 bits, the memwidth value must be set to 16.
- ☐ Because the physical width of the ROM device is 16 bits, the romwidth value must be set to 16.
- ☐ Intel format must be used.

The EPROM programmer does not require a ROM image, so the addresses in the input hex output file do not need to be contiguous.

Because memwidth and romwidth have the same value, only one output file is generated (the number of output files is determined by the ratio of memwidth to romwidth). The output file is named with the `-o` option.

A ROMS directive is used in this scenario since the `paddr` option is used to relocate both `secA` and `secB`.

The hex conversion command file for Scenario 2 is shown in Example D–7. This command file uses the following options to select the requirements of the system:

Option	Description
<code>-i</code>	Create Intel format
<code>-map example2.mxp</code>	Generate example2.mxp as the map file of the conversion
<code>-o example2.hex</code>	Name example2.hex as the output file
<code>-memwidth 8</code>	Set EPROM system memory width to 8
<code>-romwidth 8</code>	Set physical ROM width to 8

Example D–7. Hex Conversion Command File for Scenario 2

```
/* Hex Conversion Command file for Scenario 2          */
a.out          /* linked COFF object file, input */
-i            /* Intel format */

/* The following two options are optional */
-map example2.mxp /* Generate a map of the conversion */
-o example2.hex   /* Resulting Hex Output file */

/* Specify EPROM system Memory Width and Physical ROM width */
-memwidth 16      /* EPROM memory system width */
-romwidth 16      /* Physical width of ROM */

ROMS
{
    EPROM: origin = 0x0, length = 0x20000
}

SECTIONS
{
    secA: paddr = 0x0
    secB: paddr = 0x1000
}
```

Example D-8 shows the contents of the resulting map file (example2.mxp).
Example D-9 shows the contents of the resulting hex output file (example2.hex).

Example D-8. Contents of Hex Map File example2.mxp

```
*****
TMS470 COFF/Hex Converter                      Version x.xx
*****
Mon Sep 18 19:34:47 1995

INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel

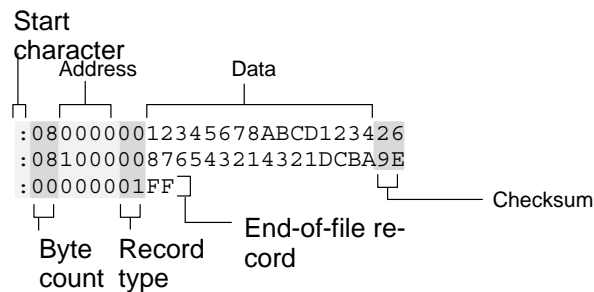
PHYSICAL MEMORY PARAMETERS
  Default data width:      8
  Default memory width:   16
  Default output width:   16

OUTPUT TRANSLATION MAP
-----
00000000..0001ffff  Page=0  ROM Width=16  Memory Width=16  "EPROM"
-----

  OUTPUT FILES: example2.hex [b0..b15]

  CONTENTS: 00000000..00000003  Data Width=1  secA
            00001000..00001003  Data Width=1  secB
```

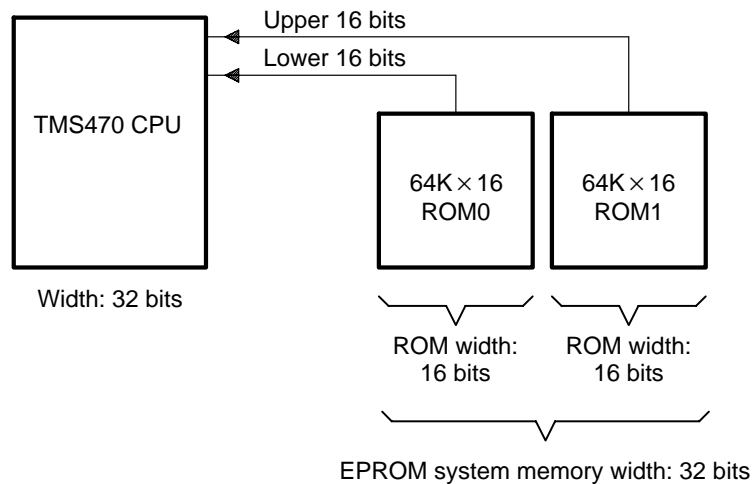
Example D-9. Contents of Hex Output File, example2.hex



D.3 Scenario 3: Building a Hex Conversion Command File for Two 8-Bit EPROMs

Scenario 3 shows how to build the hex conversion command file for converting a COFF object file for the memory system shown in Figure D–3. In this system, there are two external $64\text{K} \times 16\text{-bit}$ EPROMs interfacing with the TMS470 target processor. The application code and data will be burned on the EPROM starting at address 0x20. The application code will be burned first, followed by the data tables.

Figure D–3. EPROM Memory System for Scenario 3



In this scenario, the EPROM load address for the application code and for the data also corresponds to the TMS470 CPU address that accesses the code and data. Therefore, only a load address needs to be specified.

Example D–10 shows the linker command file for this scenario.

Example D–10. Linker Command File for Scenario 3

```
/* **** Scenario 3 Link Command **** */
/*
/* Usage:  lnk470 <obj files...>    -o <out file> -m <map file> lnk32.cmd */
/*         cl470  <src files...> -z -o <out file> -m <map file> lnk32.cmd */
/*
/* Description: This file is a sample command file that can be used
/*              for linking programs built with the TMS470 C
/*              compiler. Use it as a guideline; you may want to change
/*              the allocation scheme according to the size of your
/*              program and the memory layout of your target system.
/*
/* Notes: (1)  You must specify the directory in which rts32.lib is
/*            located. Either add a "-i<directory>" line to this
/*            file, or use the system environment variable C_DIR to
/*            specify a search path for libraries.
/*
/*            (2) If the runtime-support library you are using is not
/*              named rts32.lib, be sure to use the correct name here.
/* **** */
-m example3.map

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
    I_MEM   : org = 0x00000000    len = 0x00000020 /* INTERRUPTS          */
    D_MEM   : org = 0x00000020    len = 0x00010000 /* DATA MEMORY (RAM) */
    P_MEM   : org = 0x00010020    len = 0x00100000 /* PROGRAM MEMORY (ROM) */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    secA: load = 0x20
    secB: load = D_MEM
}
```

You must create a hex conversion command file to generate a hex output with the correct addresses and format for the EPROM programmer.

The EPROM programmer in this scenario has the following system requirements:

- ☐ In the memory system outlined in Figure D–3, the EPROM system memory width is 32 bits because each of the physical ROMs provides 16 bits of a 32-bit word. Because the EPROM system memory width is 32 bits, the memwidth value must be set to 32.
- ☐ Because the width of each of the physical ROMs is 16 bits, the romwidth value must be set to 16.
- ☐ Intel format must be used.

With a memwidth of 32 and a romwidth of 16, two output files are generated by the hex conversion utility (the number of files is determined by the ratio of memwidth to romwidth). In previous scenarios, the output filename was specified with the `-o` option. Another way to specify the output filename is to use the `files` keyword within a `ROMS` directive. When you use `-o` or the `files` keyword, the first output filename always contains the low-order bytes of the word.

The hex conversion command file for Scenario 3 is shown in Example D–11. This command file uses the following options to select the requirements of the system:

Option	Description
<code>-i</code>	Create Intel format
<code>-map example3.mxp</code>	Generate example3.mxp as the map file of the conversion
<code>-memwidth 32</code>	Set EPROM system memory width to 32
<code>-romwidth 16</code>	Set physical ROM width to 16

The `files` keyword is used within the `ROMS` directive to specify the output filenames.

Example D–11. Hex Conversion Command File for Scenario 3

```
/* Hex Conversion Command file for Scenario 3          */
a.out          /* linked COFF object file, input */
-i            /* Intel format */

/* Optional Commands */

-map example3.mxp /* Generate a map of the conversion */
/* Specify EPROM system memory width and physical ROM width */
-memwidth 32      /* EPROM memory system width */
-romwidth 16      /* Physical width of ROM */

ROMS
{
    EPROM: org = 0x0, length = 0x20000
    files={ lower16.bit, upper16.bit }
}
```

Example D–12 shows the contents of the resulting map file (example3.mxp).

Example D–12. Contents of Hex Map File example3.mxp

```
*****
TMS470 COFF/Hex Converter          Version x.xx
*****
Tue Sep 19 07:41:28 1995

INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel

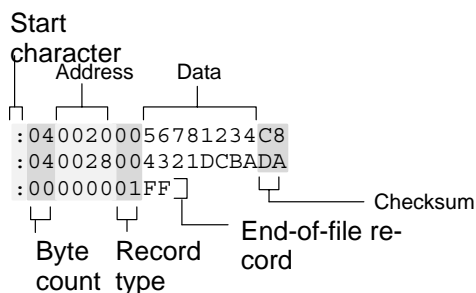
PHYSICAL MEMORY PARAMETERS
    Default data width:      8
    Default memory width:    32
    Default output width:    16

OUTPUT TRANSLATION MAP
-----
00000000..0001ffff  Page=0  ROM Width=16  Memory Width=32  "EPROM"
-----
    OUTPUT FILES: lower16.bit [b0..b15]
                  upper16.bit [b16..b31]

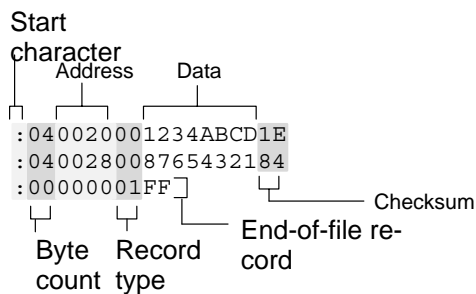
    CONTENTS: 00000020..00000021  Data Width=1  secA
              00000028..00000029  Data Width=1  secB
```

The contents of the output files lower16.bit and upper16.bit are shown in Example D–13 and Example D–14, respectively. The low-order 16 bits of the 32-bit output word are stored in the lower16.bit file, while the upper 16 bits are stored in the upper16.bit file.

Example D–13. Contents of Hex Output File lower16.bit



Example D–14. Contents of Hex Output File upper16.bit



Glossary

A

absolute address: An address that is permanently assigned to a TMS470R1x memory location.

absolute lister: A debugging tool that accepts linked 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. Without the tool, an absolute listing can be prepared with the use of many manual operations.

alignment: A process in which the linker places an output section at an address that falls on an n -bit boundary, where n is a power of 2. You can specify align with the SECTIONS linker directive.

allocation: A process in which the linker calculates the final memory addresses of output sections.

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.

ASCII: American Standard Code for Information Exchange. A standard computer code for representing and exchanging alphanumeric information.

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.).

B

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.

C

C compiler: A program that translates 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 that promotes modular programming by supporting the concept of *sections*. 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 can be used as an operand.

cross-reference lister listing: An output file produced by the cross-reference lister that lists the symbols that were defined, what file they were defined in, what reference type they are, what line they were defined on, which lines referenced them, and their assembler and linker final values. The cross-reference lister uses linked object files as input.

cross-reference listing: A listing produced 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 value as determined by the input assembly source file.

D

.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).

E

emulator: A hardware development system that emulates TMS470R1x 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 TMS470R1x 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.

F

field: For the TMS470R1x, 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).

G

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).

H

hex conversion utility: A program which 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.

I

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.

L

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 TMS470R1x 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 TMS470R1x system memory.

M

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 expansion: The source statements that are substituted for the macro call and are subsequently assembled.

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.

magic number: A COFF file header entry that identifies an object file as a module that can be executed by the TMS470R1x.

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, which 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.

N

named section: An initialized section that is defined with a `.sect` directive.

O

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.

P

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.

Q

quiet run: An option that suppresses the normal banner and the progress information.

R

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.

ROM width: The width (in bits) of each output file, or , more specifically, the width of a single data value in the file. The ROM width determines how the utility partitions the data into output files. After the target words are mapped to memory words, the memory words are broken into one or more output files. The number of output files is determined by the ROM width.

run address: The address where a section runs.

S

section: A relocatable block of code or data that will ultimately occupy contiguous space in the TMS470R1x 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: See *SPC*.

sign-extend: To fill the unused MSBs of a value with the value's sign bit.

simulator: A software development system that simulates TMS470R1x operation.

source file: A file that contains C code or assembly language code that will be compiled or assembled to form an object file.

SPC (section program counter): An element that keeps track of the current location within a section; each section has its own SPC.

static variable: A variable 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 TMS470R1x 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.

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.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

T

tag: An optional *type* name that can be assigned to a structure, union, or enumeration.

target memory: Physical memory in a TMS470R1x 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.

U

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: A kind of value that is treated as a positive number, regardless of its actual sign.

W

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 32-bit addressable location in target memory.

- . symbol for SPC 7-55
- ! operand suffix 3-13
- ? operand suffix 3-19
- ; in assembly language source 3-15
- " " in character constants 3-17
- () in expressions 3-26
- [] operand syntax 3-12
- { } operand syntax 3-14
- # operand prefix 3-12
- \$ in local label 3-19
- \$ symbol for SPC 3-23
- \$\$ in substitution symbol functions 5-7
- @ archiver command 6-4
- @ assembler option 3-4
- * in assembly language source 3-15
- /* and */ delimiters 7-21, 11-6
- ^ operand suffix 3-13

A

- a archiver command 6-4
- A operand (.option directive) 4-17
- a option
 - hex conversion utility 11-4, 11-26
 - linker 7-6
 - name utility 10-16
- A_DIR environment variable 3-8, 7-12, 7-13
- aa assembler option 3-4
- abs470 command 8-3
- absolute lister
 - abs470 command 8-3
 - creating the absolute listing file 8-2
 - definition E-1
 - description 1-4, 8-1 to 8-10
 - development flow 8-2
 - example 8-5 to 8-10
- absolute lister (continued)
 - invoking 8-3
 - options
 - e 8-3
 - q 8-3
- absolute listing 3-5
 - aa assembler option 3-4
 - producing 8-2
- absolute output module 7-6
- ac assembler option 3-4
- ahc assembler option 3-4
- ahi assembler option 3-4
- al assembler option 3-5
- .align directive 4-15, 4-24
- align keyword 7-34
- alignment 4-15 to 4-16, 4-24, 7-36
 - definition E-1
 - with padding 7-36
- allocation
 - alignment 4-24, 7-36
 - alignment with padding 7-36
 - binding 7-35
 - blocking 7-36
 - default, memory 2-12, 7-35 to 7-36
 - default algorithm 7-52 to 7-53
 - definition 2-2, E-1
 - description 7-33 to 7-43
 - GROUP 7-50
 - in .bss section 4-28
 - of output sections 7-29
 - partitioning memory, figure 2-3
 - UNION 7-48
- alternate directories
 - for assembler input 3-7 to 3-9, 7-12
 - naming with -i option 3-7
 - naming with A_DIR 3-8
- apd assembler option 3-5
- api assembler option 3-5

- ar linker option 7-7
- ar470 command 6-4
- archive libraries
 - adding new members 6-6
 - as object libraries 7-24 to 7-25
 - back referencing 7-19
 - definition E-1
 - exhaustively reading 7-19
 - modifying a member 6-6
 - naming members 6-5
 - naming with .mlib directive 4-55 to 4-56
 - naming with -l option 7-11
 - printing table of contents 6-6
 - types of files 6-2
- archiver
 - commands 6-4
 - @ 6-4
 - a 6-4
 - d 6-4
 - r 6-4
 - t 6-4
 - u 6-4
 - x 6-5
 - definition E-1
 - description 1-3, 6-1 to 6-7
 - examples 6-6
 - in the software development flow 6-3
 - invoking 6-4
 - options
 - q 6-5
 - s 6-5
 - v 6-5
- args linker option 7-8
- arguments, passing to the loader 7-8
- arithmetic operators 3-27
- as assembler option 3-5
- ASCII-Hex object format 11-1, 11-26
- .asg directive
 - description 4-21, 4-25
 - listing control 4-17, 4-36
 - use in macros 5-6
- asm470 command 3-4
- .asmfunc directive 4-23, 4-27
- assembler
 - character strings 3-18
 - constants 3-16 to 3-17
 - cross-reference listings 3-5, 3-37
 - default section directive 2-4
- assembler (continued)
 - definition E-1
 - description 1-3, 3-1 to 3-38
 - expressions 3-26 to 3-30
 - handling COFF sections 2-4 to 2-10
 - in the software development flow 3-3
 - invoking 3-4
 - macros 5-1 to 5-24
 - naming alternate directories for input 3-7
 - options
 - @ 3-4
 - aa 3-4
 - ac 3-4
 - ahc 3-4
 - ahi 3-4
 - al 3-5
 - apd 3-5
 - api 3-5
 - as 3-5
 - ax 3-5, 3-37
 - d 3-5, 3-21
 - f 3-5
 - g 3-5
 - i 3-5, 3-7
 - l 3-31
 - me 3-5
 - mt 3-6
 - q 3-6
 - u 3-6
 - output listing 3-33 to 3-34, 4-17 to 4-18
 - directive listing* 4-17, 4-36
 - enabling* 4-17, 4-52
 - false conditional block listing* 4-17, 4-40
 - list options* 4-17 to 4-18, 4-60
 - macro listing* 4-55 to 4-56, 4-57
 - page eject* 4-18, 4-62
 - page length* 4-17, 4-51
 - page width* 4-18, 4-51
 - substitution symbol listing* 4-66
 - suppressing* 4-17, 4-52
 - tab size* 4-18, 4-72
 - title* 4-18, 4-74
 - overview 3-2
 - relocation 2-13 to 2-14, 7-6 to 7-7
 - at run time* 2-15
 - sections directives 2-4 to 2-10
 - source listings 3-31 to 3-34
 - source statement format 3-10 to 3-15
 - symbols 3-19 to 3-25

- assembler directives 4-1 to 4-78
 - absolute lister 8-7
 - aligning the section program counter (SPC),
.align 4-15, 4-24
 - changing the instruction type
.state16 4-10, 4-67
.state32 4-10, 4-68
 - default directive 2-4
 - defining assembly-time symbols 4-21 to 4-22
.asg 4-21, 4-25
.endstruct 4-22, 4-70
.equ 4-21, 4-63
.eval 4-21, 4-25
.label 4-21, 4-50
.set 4-21, 4-63
.struct 4-22, 4-70
.tag 4-22, 4-70
 - defining sections 4-7 to 4-9
.bss 2-4, 4-7, 4-28
.data 2-4, 4-7, 4-34
.sect 2-4, 4-7, 4-62
.text 2-4, 4-7, 4-73
.usect 2-4, 4-7, 4-76
 - enabling conditional assembly 4-20
.break 4-20, 4-53
.else 4-20, 4-47
.elseif 4-20, 4-47
.endif 4-20, 4-47
.endloop 4-20, 4-53
.if 4-20, 4-47
.loop 4-20, 4-53
 - example 2-8 to 2-10
 - formatting the output listing 4-17 to 4-18
.drlist 4-17, 4-36
.drnolist 4-17, 4-36
.fclist 4-17, 4-40
.fcnolist 4-17, 4-40
.length 4-17, 4-51
.list 4-17, 4-52
.mlist 4-17, 4-57
.mnolist 4-17, 4-57
.nolist 4-17, 4-52
.option 4-17 to 4-18, 4-60
.page 4-18, 4-62
.sslist 4-18, 4-66
.ssnolist 4-18, 4-66
.tab 4-18, 4-72
.title 4-18, 4-74
.width 4-18, 4-51
- assembler directives (continued)
 - initializing constants 4-11 to 4-14
.bes 4-11, 4-64
.byte 4-11, 4-30
.char 4-11, 4-30
.double 4-11, 4-35
.field 4-12, 4-41
.float 4-12, 4-43
.half 4-12, 4-46
.int 4-13, 4-49
.long 4-13, 4-49
.short 4-12, 4-46
.space 4-11, 4-64
.string 4-13, 4-69
.word 4-13, 4-49
 - miscellaneous directives 4-23
.asmfunc 4-27
.asmfunc 4-23
.clink 4-7, 4-31
.emsg 4-23, 4-37
.end 4-23, 4-39
.endasmfunc 4-23, 4-27
.mmsg 4-23, 4-37
.newblock 4-23, 4-59
.wmsg 4-23, 4-37
 - reference 4-24 to 4-78
 - referencing other files 4-19
.copy 4-19, 4-32
.def 4-19, 4-44
.global 4-19, 4-44
.include 4-19, 4-32
.mlib 4-19, 4-55
.ref 4-19, 4-44
 - summary table 4-2 to 4-6
- assembly language development flow 1-2, 3-3, 6-3, 7-3
- assembly-time constants 3-17, 4-63, E-1
- assigning a value to a symbol 4-63
- assignment expressions 7-55 to 7-56
- assignment statement syntax 7-54
- attributes 3-38, 7-28
- autoinitialization
 - at load time 7-9, 7-85
described 7-85
 - at run time 7-9, 7-84
 - definition E-2
- auxiliary entries A-16, E-2
- ax assembler option 3-5, 3-37

B

- b linker option 7-8
- B operand (.option directive) 4-17
- .bes directive 4-11, 4-64
- big endian, defined E-2
- binary integer constants 3-16
- binding 7-34, 7-35
 - definition E-2
- .block COFF debugging directive B-3
- block keyword 7-34
- blocking 7-36
- boot option, hex conversion utility 11-4
- boot.obj module 7-83, 7-86
- boot-time copy table generated by linker 7-70 to 7-71
- .break directive
 - description 4-20, 4-53
 - listing control 4-17, 4-36
 - use in macros 5-14 to 5-15
- .bss directive 2-4, 4-7, 4-28
- .bss section 4-7, 4-28, A-3
 - definition E-2
 - holes 7-64
 - initializing 7-64
- .bss symbol defined by linker 7-57
- .byte directive
 - description 4-11, 4-30
 - limiting listing with the .option directive 4-17, 4-60
- byte option, hex conversion utility 11-4

C

- C code, linking 7-83 to 7-86
 - example 7-87 to 7-89
- C compiler
 - and COFF object file structure A-1
 - and linking conventions 7-9
 - definition E-2
 - description 1-3
 - special symbols A-13
 - storage classes A-15
 - symbol table entries A-13
 - symbolic debugging A-1
- C language linker options 7-9

- C memory pool 7-11, 7-84
- c option
 - linker 7-9, 7-57
 - name utility 10-16
- C system stack 7-16, 7-84
- C/C++ compiler, symbolic debugging directives B-1 to B-14
- C_DIR environment variable 7-11, 7-12 to 7-13
- _c_int00 7-9, 7-86
- .char directive
 - description 4-11, 4-30
 - limiting listing with the .option directive 4-17, 4-60
- character constants 3-17
- character strings 3-18
- .cinit section 7-84
- cinit symbol 7-84
- .cinit tables 7-84
- .clink directive 4-7, 4-31
- COFF
 - and the linker 7-1
 - auxiliary entries A-16
 - conversion to hexadecimal format 11-1 to 11-30
 - default allocation 7-52 to 7-53
 - description A-1 to A-16
 - file headers A-4 to A-5
 - file structure A-2 to A-3
 - initialized sections 2-2, 2-6
 - introduction 2-1 to 2-18
 - loading a program 2-16
 - object file example A-3
 - optional file header A-6
 - relocation 2-13 to 2-14, A-10 to A-11
 - relocation type* A-11
 - run-time relocation* 2-15
 - symbol table index* A-10
 - virtual address* A-10
 - sample COFF object file A-3
 - section headers A-7 to A-9
 - sections 2-2 to 2-3
 - allocation* 2-2
 - assembler* 2-4 to 2-10
 - description* 2-2
 - initialized* 2-2, 2-6
 - linker* 2-11 to 2-13
 - named* 2-6 to 2-7, 7-62
 - special types* 7-51
 - uninitialized* 2-2, 2-4 to 2-5
 - special symbols A-13

- COFF (continued)
 - storage classes A-15
 - string table A-14
 - structuring the information A-10
 - symbol table 2-17 to 2-18
 - format* A-14
 - special symbols* A-13
 - storage classes* A-15
 - string table* A-14
 - symbol values* A-15
 - symbolic debugging directives B-3
 - syntax* B-4
 - uninitialized sections 2-4 to 2-5, A-2
- command files
 - appending to command line 3-4
 - definition E-2
 - hex conversion utility 11-6 to 11-7
 - linker 7-4, 7-21 to 7-23
 - constants in* 7-23
 - example* 7-88
 - reserved words* 7-23
- comment field, in assembler syntax 3-15
- comments
 - definition E-2
 - in a linker command file 7-21
 - in assembly language source code 3-15
 - in directive syntaxes 4-2
 - in macros 5-17
 - that extend past page width 4-51
- common object file format, defined E-2
- conditional blocks 4-47, 5-14 to 5-15
 - assembly directives 4-20, 4-47
 - in macros* 5-14 to 5-15
 - maximum nesting levels* 5-14 to 5-15
 - listing of false conditional blocks 4-40
- conditional expressions 3-28
- conditional linking 4-31, 7-13
- conditional processing, definition E-2
- configured memory 7-53
 - definition E-3
- constants 3-16 to 3-17, 3-21
 - assembly-time 3-17, 4-63
 - binary integers 3-16
 - character 3-17
 - decimal integers 3-16
 - definition E-3
 - floating-point 4-35, 4-43
 - hexadecimal integers 3-17
- constants (continued)
 - in linker command files 7-23
 - octal integers 3-16
 - symbolic 3-23
 - \$* 3-23
 - coprocessor IDs* 3-23
 - defining (–d assembler option)* 3-21
 - predefined* 3-23
 - processor symbols* 3-24
 - register symbols* 3-23
 - status registers* 3-24
 - symbols as 3-21
- coprocessor IDs 3-23
- .copy directive 3-7, 4-19, 4-32
- copy files 3-7, 4-32
 - ahc assembler option 3-4
- copy routine, general-purpose 7-73 to 7-76
- COPY section 7-51
- copy tables automatically generated by linker 7-69 to 7-70
 - contents 7-72 to 7-73
 - sections and symbols 7-77 to 7-78
- cr linker option 7-9, 7-57, 7-85
- creating holes 7-62 to 7-64
- cross-reference lister
 - creating the cross-reference listing 9-2
 - description 1-4, 9-1 to 9-6
 - development flow 9-2
 - example 9-4
 - invoking 9-3
 - listing, definition E-3
 - listings 3-5
 - options 9-3
 - reference types of symbols 9-5
 - symbol attributes 9-5
 - xref470 command 9-3
- cross-reference listing
 - definition E-3
 - description 3-37 to 3-38
 - producing with the .option assembler directive 4-18 to 4-19, 4-60
- CSPR register 3-24

D

- d archiver command 6-4
- d assembler option 3-5, 3-21
- d option, name utility 10-16
- .data directive 2-4, 4-7, 4-34

- `.data` section 4-7, 4-34, A-3
 - definition E-3
- `.data` symbol defined by linker 7-57
- `-datawidth`
 - option, hex conversion utility 11-4
- decimal integer constants 3-16
- `.def` directive 2-17, 4-19, 4-44
- default
 - allocation 7-52 to 7-53
 - fill value for holes 7-10
 - memory allocation 2-12
 - MEMORY configuration 7-52 to 7-53
 - MEMORY model 7-26
 - SECTIONS configuration 7-30, 7-52 to 7-53
- defining macros 5-3 to 5-4
- development flow diagram 1-2
- development tools overview 1-2
- directives, definition E-3
- directory search algorithm
 - assembler 3-7 to 3-9
 - linker 7-11
- disabling conditional linking (`-j` option) 7-13
- `.double` directive 4-11, 4-35
- `.drlist` directive
 - description 4-17, 4-36
 - use in macros 5-20
- `.drnolist` directive
 - description 4-17, 4-36
 - use in macros 5-20
- DSECT section 7-51
- dummy section 7-51
- DWARF debugging directives B-2
- DWARF symbolic debugging directives, syntax B-4
- `.dwattr` DWARF debugging directive B-2
- `.dwcfi` DWARF debugging directive B-2
- `.dwcie` DWARF debugging directive B-2
- `.dwendentry` DWARF debugging directive B-2
- `.dwendtag` DWARF debugging directive B-2
- `.dwfde` DWARF debugging directive B-2
- `.dwpsn` DWARF debugging directive B-2
- `.dwtag` DWARF debugging directive B-2

E

- `-e` option
 - absolute lister 8-3
 - linker 7-9
- `edata` linker symbol 7-57
- EDEF symbol reference type 9-5
- `.else` directive
 - description 4-20, 4-47
 - use in macros 5-14 to 5-15
- `.elseif` directive
 - description 4-20, 4-47
 - use in macros 5-14 to 5-15
- `.emsg` directive
 - description 4-23, 4-37
 - listing control 4-17, 4-36
 - use in macros 5-17
- `.end` directive 4-23, 4-39
- end linker symbol 7-57
- END() linker operator 7-59
- `.endasmfunc` directive 4-23, 4-27
- `.endblock` COFF debugging directive B-3
- `.endfunc` COFF debugging directive B-3
- `.endif` directive
 - description 4-20, 4-47
 - use in macros 5-14 to 5-15
- `.endloop` directive
 - description 4-20, 4-53
 - use in macros 5-14 to 5-15
- `.endm` directive 5-3
- `.endstruct` directive 4-70
 - description 4-22
- entry points 7-9
 - `_c_int00` 7-9, 7-86
 - default value 7-9
 - definition E-3
 - for C code 7-86
 - for the linker (`-e` option) 7-9
 - `_main` 7-9
- environment variables
 - A_DIR 3-8, 7-12
 - C_DIR 7-11 to 7-13
- `.eos` COFF debugging directive B-3
- EPROM programmer 1-4
- `.equ` directive 4-21, 4-63
- EREF symbol reference type 9-5

error messages
 generating 4-23
 producing in macros 5-17

.etag COFF debugging directive B-3

etext linker symbol 7-57

.eval directive
 description 4-21, 4-25
 listing control 4-17, 4-36
 use in macros 5-7

executable module, definition E-3

executable output module 7-6, 7-7

executable relocatable output module 7-7

explicit initialization of uninitialized sections 7-65

expressions 3-26 to 3-30
 absolute and relocatable 3-28 to 3-30
 examples 3-29 to 3-30
 arithmetic operators 3-27
 conditional 3-28
 conditional operators 3-28
 definition E-3
 illegal 3-29
 left-to-right evaluation 3-26
 linker 7-55 to 7-56
 operators 3-27
 overflow 3-27
 parentheses' effect on evaluation 3-26
 precedence of operators 3-26
 relocatable symbols in 3-28 to 3-30
 underflow 3-27
 well-defined 3-28

Extended Tektronix object format 11-1, 11-30

external symbols 2-17, 3-28, 4-44
 definition E-3

F

-f linker option 7-10

-f option
 assembler 3-5
 name utility 10-16

far calls 7-16

.fclist directive
 description 4-17, 4-40
 listing control 4-17, 4-36
 use in macros 5-19

.fcnolist directive
 description 4-17, 4-40
 listing control 4-17, 4-36
 use in macros 5-19

.field directive 4-12, 4-41

file
 copy 3-4
 include 3-4

.file COFF debugging directive B-3

file headers A-4 to A-5
 definition E-4

filenames
 as character strings 3-18
 copy/include files 3-7
 extensions, changing defaults 8-3
 macros, in macro libraries 5-13

files ROMS specification 11-14

-fill hex conversion utility option 11-4, 11-23

fill MEMORY specification 7-28

fill ROMS specification 11-14

fill value 7-64 to 7-65
 default for linker (-f option) 7-10
 setting 7-10

filling holes 7-64 to 7-65

filling memory ranges 7-28

.float directive 4-12, 4-43

floating-point constants 4-35, 4-43

forced substitution 5-9

format of assembler source statements 3-10 to 3-14

.func COFF debugging directive B-3

G

-g option
 assembler 3-5
 linker 7-10
 name utility 10-16
 object file display 10-2

.global directive 2-17, 4-19, 4-44

global symbols 7-10
 definition E-4
 making static with -h option 7-10
 overriding -h option 7-10

GROUP linker directive 7-50, E-4

H

-h linker option 7-10

- H operand (.option directive) 4-17
 - h option, name utility 10-16
 - .half directive
 - description 4-12, 4-46
 - limiting listing with the .option directive 4-17, 4-60
 - heap linker option 7-11, 7-84
 - heap size 7-11
 - hex conversion utility
 - command files 11-6 to 11-7
 - invoking 11-3, 11-6
 - ROMS directive 11-6
 - SECTIONS directive 11-6
 - configuring memory widths
 - defining memory word width (memwidth) 11-4
 - specifying output width (romwidth) 11-4
 - definition E-4
 - description 1-4, 11-1 to 11-30
 - examples D-1 to D-16
 - avoiding holes with multiple sections D-2 to D-7
 - building a hex command file for 16-BIS code D-8 to D-11
 - building a hex command file for two 8-bit EPROMS D-12 to D-16
 - formats supported 11-1
 - generating a map file 11-4
 - generating a quiet run 11-4
 - hex470 command 11-3
 - image mode
 - defining the target memory 11-23
 - filling holes 11-4, 11-23
 - invoking 11-4, 11-22
 - resetting address origin 11-4, 11-24
 - in the development flow 11-2
 - indicating big endian 11-4
 - indicating little endian 11-4
 - invoking 11-3 to 11-7
 - from the command line 11-3
 - in a command file 11-3
 - memory width (memwidth) 11-9 to 11-10
 - exceptions 11-9
 - numbering output locations by bytes 11-4
 - options
 - a 11-26
 - fill 11-23
 - i 11-27
 - image 11-22
 - m 11-28
 - hex conversion utility, options (continued)
 - map 11-17 to 11-18
 - memwidth 11-9
 - o 11-21
 - romwidth 11-11
 - summary table 11-4
 - t 11-29
 - x 11-30
 - output filenames 11-4, 11-20
 - default filenames 11-21
 - ROMS directive 11-7
 - representing the width of data words 11-4
 - ROM width (romwidth) 11-10 to 11-15
 - ROMS directive 11-13 to 11-18
 - creating a map file of 11-17 to 11-30
 - defining the target memory 11-23
 - example 11-15 to 11-18
 - parameters 11-13 to 11-14
 - specifying output filenames 11-7
 - SECTIONS directive 11-19 to 11-20
 - parameters 11-19 to 11-20
 - selecting boot mode 11-4
 - setting the
 - of data items 11-4
 - target width 11-9
 - zero option 11-24
 - hex470 command 11-3
 - hexadecimal integer constants 3-17
 - holes 7-10, 7-62 to 7-65
 - creating 7-62 to 7-64
 - definition E-4
 - fill value 7-31, 11-14, 11-23
 - filling 7-64 to 7-65, 11-23
 - in output sections 7-62 to 7-65
 - in uninitialized sections 7-65
- I
- I MEMORY attribute 7-28
 - i option
 - assembler 3-5
 - examples by operating system 3-8
 - maximum number per invocation 3-7
 - hex conversion utility 11-4, 11-27
 - linker 7-12
 - identifying external symbols 2-17
 - .if directive
 - description 4-20, 4-47
 - use in macros 5-14 to 5-15

- image hex conversion utility option 11-4, 11-22
- immediate value as operands 3-15
- immediate value operand prefix (#) 3-12
- .include directive 3-7, 4-19, 4-32
- include files 3-4, 3-7, 4-32
- incremental linking 7-81 to 7-82
 - definition E-4
- indirect address, operand syntax ([]) 3-12
- initialization at run time 7-84
- initialized sections 2-6, 7-62
 - .data section 2-6, 4-34
 - definition E-4
 - .sect section 2-6, 4-62
 - .text section 2-6, 4-73
- input
 - linker 7-3, 7-24 to 7-25
 - sections 7-38 to 7-40, E-4
- instruction set, selecting
 - .state16 directive 4-10, 4-67
 - .state32 directive 4-10, 4-68
- .int directive
 - description 4-13, 4-49
 - limiting listing with the .option directive 4-17 to 4-18, 4-60
- Intel MCS-86 object format 11-1, 11-27
- invoking
 - absolute lister 8-3
 - archiver 6-4
 - assembler 3-4
 - cross-reference lister 9-3
 - hex conversion utility 11-3 to 11-7
 - linker 7-4
 - name utility 10-16
 - object file display utility 10-2
 - strip utility 10-17

J

- j linker option 7-13

K

- keywords
 - allocation parameters 7-34
 - load 2-15, 7-34, 7-44
 - run 2-15, 7-34, 7-44 to 7-46

L

- l option
 - assembler, source listing format 3-31
 - cross-reference lister 9-3
 - linker 7-11
 - name utility 10-16
- label, case sensitivity 3-4
- .label directive 4-21, 4-50
- label field v, 3-10 to 3-11
- labels 3-19
 - defined and referenced (cross-reference list) 3-37
 - definition E-5
 - in assembly language source 3-10 to 3-11
 - local 3-19, 4-59
 - symbols used as 3-19 to 3-20
 - syntax 3-10 to 3-11
 - using with .byte directive 4-30
- left-to-right evaluation (of expressions) 3-26
- .length directive
 - description 4-17, 4-51
 - listing control 4-17
- length MEMORY specification 7-28
- length ROMS specification 11-14
- libraries, object 7-24
- library search, using alternate mechanism, –priority
 - linker option 7-19
- library search algorithm 7-11 to 7-13
- library-build utility 1-4
- .line COFF debugging directive B-3
- line-number table, line-number entries, definition E-5
- linker
 - ar option 7-7
 - | operator 7-40
 - allocation to multiple memory ranges 7-40
 - assigning symbols 7-54
 - assignment expressions 7-54, 7-55 to 7-56
 - automatic splitting of output sections 7-41
 - >> operator 7-41
 - C code 7-83 to 7-86
 - COFF 7-1
 - command file 7-4, 7-21 to 7-23
 - command file example 7-88
 - configured memory 7-53
 - default memory allocation 2-12
 - definition E-5

linker (continued)

- description 1-3, 7-1 to 7-90
- END() operator 7-59
- example 7-87 to 7-90
- generated copy tables. See linker-generated copy tables
- GROUP statement 7-48, 7-50
- handling COFF sections 2-11 to 2-13
- in the software development flow 7-3
- input 7-3, 7-21 to 7-23
- invoking 7-4
- keywords 7-23, 7-44 to 7-47
- linking C code 7-9, 7-83 to 7-86
- Ink470 command 7-4
- LOAD_END() operator 7-59
- LOAD_SIZE() operator 7-59
- LOAD_START() operator 7-59
- loading a program 2-16
- MEMORY directive 2-11, 7-26 to 7-29
- object libraries 7-24 to 7-25
- operators 7-56
- options
 - a 7-6
 - ar 7-7
 - args 7-8
 - b 7-8
 - c 7-9
 - cr 7-9, 7-85
 - e 7-9
 - f 7-10
 - g 7-10
 - h 7-10
 - heap 7-11
 - i 7-12
 - j 7-13
 - l 7-11
 - m 7-13 to 7-15
 - o 7-15
 - r 7-7
 - s 7-15
 - stack 7-16
 - summary table 7-5
 - trampolines 7-16
 - u 7-18
 - w 7-19
 - x 7-19
- output 7-3, 7-15, 7-87
- overview 7-2
- partial linking 7-81 to 7-82
- RUN_END() operator 7-59

linker (continued)

- RUN_SIZE() operator 7-59
- RUN_START() operator 7-59
- section run-time address 7-44 to 7-47
- sections 2-13
 - output 7-52
 - special 7-51
- SECTIONS directive 2-11, 7-30 to 7-43
- SIZE() operator 7-59
- START() operator 7-59
- symbols 2-17 to 2-18, 7-57
- table() operator 7-70, 7-71
- unconfigured memory, overlaying 7-51
- UNION statement 7-48 to 7-49, 7-68

linker directives

- MEMORY 2-11, 7-26 to 7-29
- SECTIONS 2-11, 7-30 to 7-43

linker-generated copy tables 7-66 to 7-80

- automatic 7-69 to 7-70
- boot-loaded application process 7-66
 - alternative approach 7-67
- boot-time copy table 7-70 to 7-71
- contents 7-72 to 7-73
- general-purpose copy routine 7-73 to 7-76
- overlay management 7-78 to 7-80
- overlay management example 7-68
- sections and symbols 7-77 to 7-78
- splitting object components 7-78 to 7-80
- table() operator 7-70
 - manage object components 7-71

.list directive 4-17, 4-52

lister

- absolute 8-1 to 8-10
- cross-reference 9-1 to 9-6

listing

- control 4-52, 4-57, 4-60, 4-62, 4-74
- cross-reference listing 4-18, 4-60
- file 4-17 to 4-18
 - creating with the -al option 3-5
 - definition E-5
 - format 3-31 to 3-34
- page eject 4-18
- page size 4-17, 4-51

little endian, defined E-5

little-endian object code, producing with -me option 3-5

Ink470 command 7-4

load address of a section 7-44 to 7-45

- referring to with a label 7-45 to 7-47

load linker keyword 2-15, 7-44 to 7-45
 LOAD_END() linker operator 7-59
 LOAD_SIZE() linker operator 7-59
 LOAD_START() linker operator 7-59, 7-67
 loader definition E-5
 loading a program 2-16
 local labels 3-19
 logical operators 3-27
 .long directive
 description 4-13, 4-49
 limiting listing with the .option directive 4-17 to 4-18, 4-60
 .loop directive
 description 4-20, 4-53
 use in macros 5-14 to 5-15

M

M operand (.option directive) 4-17
 -m option
 hex conversion utility 11-4, 11-28
 linker 7-13 to 7-15
 -m1 option, hex conversion utility 11-5, 11-28
 -m2 option, hex conversion utility 11-5, 11-28
 -m3 option, hex conversion utility 11-5
 .macro directive 4-55, 5-3 to 5-4
 summary table 5-23 to 5-24
 macro language 5-1
 comments in 5-4
 conditional assembly 5-14 to 5-15
 defining a macro 5-3 to 5-4
 definitions of terms
 macro E-5
 macro call E-5
 macro expansion E-5
 macro library E-5
 description 5-2
 directives summary 5-23 to 5-24
 formatting the output listing 5-19 to 5-20
 labels 5-16
 nested macros 5-21 to 5-22
 parameters 5-5 to 5-12
 producing messages 5-17
 recursive macros 5-21 to 5-22
 using a macro 5-2
 varying number of arguments 5-5
 macro libraries
 accessing (.mlib directive) 5-13
 creating 5-13, 6-2
 definition E-5
 description 5-13, 6-2
 how the assembler expands a library entry 5-13
 listing control of expansions (.mllib directive) 4-56, 5-13
 using 5-13, 6-2
 macros
 comments in 5-17
 definitions of terms, macro definition E-5
 disabling macro expansion listing 4-17, 4-60
 magic number
 definition E-5
 for TMS470R1x A-4
 _main 7-9
 malloc() function 7-11, 7-84
 map file
 definition E-5
 example 7-89, 11-18
 hex conversion utility description 11-17 to 11-18
 linker description 7-13 to 7-15
 -map hex conversion utility option 11-4, 11-17
 -me assembler option 3-5
 .member COFF debugging directive B-3
 memory
 allocation 2-12, 7-35, 7-52 to 7-53
 default allocation 2-12
 map 2-13, E-6
 model 7-26
 named 7-35 to 7-36
 partitioning into logical blocks 2-3
 pool, C language 7-11, 7-84
 unconfigured 7-26
 MEMORY directive 2-11, 7-26 to 7-29
 default model 7-26, 7-52 to 7-53
 syntax 7-26 to 7-29
 memory ranges, allocation to multiple 7-40
 memory widths
 memory width (memwidth) 11-9 to 11-10
 exceptions 11-9
 ROM width (romwidth) 11-10 to 11-15
 target width 11-9
 -memwidth hex conversion utility option 11-4, 11-9
 memwidth ROMS specification 11-14
 .mexit directive 5-3

- `.mlib` directive
 - description 4-19, 4-55 to 4-56, 5-13
 - use in macros 5-13
- `.mlist` directive
 - description 4-17, 4-57
 - listing control 4-17, 4-36
 - use in macros 5-19
- `.mmsg` directive
 - description 4-23, 4-37
 - listing control 4-17, 4-36
 - use in macros 5-17
- mnemonic field
 - description 3-11
 - in syntax 3-10
- `.mnolist` directive
 - description 4-17, 4-57
 - listing control 4-17, 4-36
 - use in macros 5-19
- Motorola-S (Exorciser) object format 11-1, 11-28
- `-mt` assembler option 3-6

N

- N operand (`.option` directive) 4-17
- `-n` option, name utility 10-16
- name MEMORY specification 7-27
- name utility
 - invoking 10-16
 - options 10-16
- named memory 7-35 to 7-36
- named sections 2-6 to 2-7, A-3
 - definition E-6
 - `.sect` directive 2-7, 4-62
 - `.usect` directive 2-7, 4-76
- naming an output module (`-o` linker option) 7-15
- nested macros 5-21
- `.newblock` directive 4-23, 4-59
- `nm470` utility, invoking 10-16
- `nm55` command 10-16
- `.nolist` directive 4-17, 4-52
- NOLOAD section 7-51
- notational conventions iv

O

- O operand (`.option` directive) 4-17

- `-o` option
 - hex conversion utility 11-4
 - linker 7-15
 - name utility 10-16
 - object file display 10-2
- object code (source listing) 3-32
- object file, definition E-6
- object file display utility
 - invoking 10-2
 - options
 - `-g` 10-2
 - `-o` 10-2
 - `-x` 10-2
 - XML example application 10-9
 - XML tags generated 10-3 to 10-8
- object formats
 - address bits 11-25
 - ASCII-Hex 11-1, 11-26
 - selecting* 11-4
 - Intel MCS-86 11-1, 11-27
 - selecting* 11-4
 - Motorola-S (Exorciser) 11-1, 11-28
 - S1 records preference* 11-5, 11-28
 - S2 records preference* 11-5, 11-28
 - S3 records preference* 11-5, 11-28
 - selecting* 11-4
 - output width 11-25
 - Tektronix 11-1, 11-30
 - selecting* 11-5
 - TI-Tagged 11-1, 11-29
 - selecting* 11-5
- object libraries 7-11 to 7-13, 7-24 to 7-25
 - definition E-6
 - run-time support 7-83
 - using the archiver to build 6-2
- octal integer constants 3-16
- `ofd470` command 10-2
- `-olength`
 - option, hex conversion utility 11-4
- operands
 - definition E-6
 - field 3-12 to 3-15
 - immediate values 3-15
 - in assembler statement syntax 3-12
 - label 3-19
 - local label 3-19
 - prefixes 3-12
 - source statement format 3-12 to 3-15
 - syntaxes of 3-12 to 3-14

operator precedence order 3-27

.option directive 4-17 to 4-18, 4-60

optional file header A-6
definition E-6

options
absolute lister 8-3
archiver 6-4
assembler 3-4, 10-2
cross-reference lister 9-3
definition E-6
hex conversion utility 11-3 to 11-4
linker 7-5 to 7-20
name utility 10-16
strip utility 10-17

–order LS option, hex conversion utility 11-4

–order MS option, hex conversion utility 11-4

origin MEMORY specification 7-28

origin ROMS specification 11-13

output
absolute lister 8-3
archive library 6-5
assembler 3-1
format the listing file 4-17 to 4-18
cross-reference lister 9-3
executable 7-6
relocatable 7-7
hex conversion utility 11-4, 11-21
linker 7-3, 7-15, 7-87
module, definition E-6
module name (–o linker option) 7-15
partitioning data 11-6
sections
allocation 7-33 to 7-43
definition E-6
displaying a message 7-19
methods 7-52 to 7-53
splitting 7-41

overflow (in expression) 3-27

overlying sections 7-48 to 7-49
managing linker-generated copy tables 7-78 to 7-80

P

–p option
strip utility 10-17
name utility 10-16

page
eject 4-62
length 4-51
title 4-74
width 4-51

.page directive 4-18, 4-62

parentheses in expressions 3-26

partial linking 7-81 to 7-82
definition E-6

partitioning data into output files 11-10

placing sections in memory map 2-13

postindex indirect address 3-12

precedence groups
assembler 3-26
linker 7-56

predefined names
–d assembler option 3-5
undefining with –u assembler option 3-6

prefixes for operands 3-12

pre-index indirect address 3-12

–priority linker option 7-19

processor symbols 3-24

Q

–q option
absolute lister 8-3
archiver 6-5
assembler 3-6
cross-reference lister 9-3
hex conversion utility 11-4
name utility 10-16

quiet run
absolute lister 8-3
archiver 6-5
assembler 3-6
cross-reference lister 9-3
definition E-7
hex conversion utility 11-6

R

r archiver command 6-4

R MEMORY attribute 7-28

R operand (.option directive) 4-17

–r option
linker 7-7, 7-81 to 7-82
name utility 10-16

- recursive macros 5-21
- recursive substitution symbols 5-9 to 5-11
- .ref directive 2-17, 4-19, 4-44
- reference types for symbols 9-5
- register list in operand syntax ({ }) 3-14
- register symbols 3-23
 - 32-BIS registers 3-23
 - coprocessor registers 3-23
- related documentation vi
- relational operators in conditional expressions 3-28
- relocatable output module 7-7
- relocatable symbols 3-28 to 3-30
- relocation 3-17
 - at run time 2-15
 - capabilities 7-6 to 7-7
 - definition E-7
 - description 2-13 to 2-14
 - information A-10 to A-11
 - using -a and -r linker options 7-6
- reserved words for linker 7-23
- resetting local labels 4-59
- ROM device address 11-24
- ROM width (romwidth) 11-10 to 11-15
 - definition E-7
- romname ROMS specification 11-13
- ROMS directive 11-13 to 11-18
 - creating map file of 11-17 to 11-18
 - example 11-15 to 11-18
 - parameters 11-13 to 11-14
 - syntax 11-13
 - using image option 11-15
 - when to use 11-15
- romwidth hex conversion utility option 11-4, 11-11
- romwidth ROMS specification 11-14
- rts16.lib 7-83, 7-86
- rts32.lib 7-83, 7-86
- run address of a section 7-44 to 7-45
- run linker keyword 2-15, 7-44 to 7-45
- run-time
 - address 7-44
 - autoinitialization of variables 7-84
 - initialization 7-83
 - object libraries 7-83
 - relocation of sections 2-15
 - support 7-83

- run-time-support library 7-86
- RUN_END() linker operator 7-59
- RUN_SIZE() linker operator 7-59
- RUN_START() linker operator 7-59, 7-67

S

- s option
 - archiver 6-5
 - linker 7-15, 7-81
- search libraries
 - using -priority linker option 7-19
 - using alternate mechanism 7-19
- .sect directive 2-4, 4-7, 4-62
- .sect section 4-7, 4-62
- section
 - definition E-7
 - directives, default 2-4
 - header A-7 to A-9
 - definition E-7
 - number A-16
 - specification 7-31
- sections
 - allocation into memory 7-33 to 7-37, 7-52 to 7-53
 - COFF 2-2 to 2-3
 - creating your own 2-6 to 2-7
 - default allocation 7-52 to 7-53
 - description 2-1
 - initialized 2-6
 - input sections 7-31
 - named 2-2, 2-6 to 2-7
 - overlying with UNION directive 7-48 to 7-49
 - placing in memory map 2-13
 - relocation 2-13
 - at run time 2-15
 - special types 7-51
 - specifying a run-time address 7-44 to 7-45
 - specifying linker input sections 7-38 to 7-40
 - specifying load and run addresses 7-44 to 7-45
 - uninitialized 2-4 to 2-5
 - initializing 7-65
 - specifying a run address 7-45
- SECTIONS directive 2-11, 7-30 to 7-43
 - alignment 7-36
 - alignment with padding 7-36
 - allocation 7-33 to 7-43
 - binding 7-35
 - blocking 7-36

- SECTIONS directive (continued)
 - default, allocation 7-52 to 7-53
 - fill value 7-31
 - GROUP 7-50
 - input sections 7-31, 7-38 to 7-40
 - .label directive 7-45 to 7-47
 - load allocation 7-31
 - memory 7-35 to 7-36
 - named memory 7-35 to 7-36
 - reserved words 7-23
 - run allocation 7-31
 - section specification 7-31
 - section type 7-31
 - specifying
 - run-time address* 2-15, 7-44 to 7-47
 - two addresses* 2-15, 7-44 to 7-45
 - syntax 7-30 to 7-31
 - uninitialized sections 7-45
 - UNION 7-48 to 7-50
 - use with MEMORY directive 7-26
- SECTIONS hex conversion utility directive 11-19 to 11-20
 - parameters 11-19 to 11-20
- SECTIONS linker directive
 - allocation using multiple memory ranges 7-40
 - splitting of output sections 7-41
- .set directive 4-21, 4-63
- set S bit operand suffix (^) 3-13
- .setsect absolute lister directive 8-7
- .setsym absolute lister directive 8-7
- shifted registers in operand syntax 3-14
- .short directive
 - description 4-12, 4-46
 - limiting listing with the .option directive 4-17, 4-60
- sign-extend definition E-7
- 16-bit instructions, use by default (–mt option) 3-6
- SIZE() linker operator 7-59, 7-67
- sname SECTIONS specification 11-19
- software development
 - absolute lister in 8-2
 - archiver in 6-3
 - assembler in 3-3
 - cross-reference lister in 9-2
 - flow diagram 1-2
 - hex conversion utility in 11-2
 - linker in 7-3
 - tools overview 1-2
- source file
 - assembler 10-2
 - definition E-7
- source listings 3-31 to 3-34
- source statement
 - field (source listing) 3-32
 - format 3-10 to 3-15
 - comment field* 3-15
 - label field* 3-10 to 3-11
 - mnemonic field* 3-11
 - operand field* 3-12 to 3-15
 - number (source listing) 3-31 to 3-34
- .space directive 4-11, 4-64
- SPC 2-8
 - aligning
 - by creating a hole* 7-62
 - to word boundaries* 4-15 to 4-16, 4-24
 - assembler's effect on 2-8 to 2-10
 - assigning a label to 3-11
 - assigning to a label 3-11
 - definition E-7
 - linker symbol 7-55, 7-62
 - maximum number of 2-8
 - predefined symbol for 3-23
 - value
 - associated with labels* 3-11
 - shown in source listings* 3-31
- special section types 7-51
- special symbols in the symbol table A-13
- SPSR register 3-24
- square brackets in operand syntax 3-12
- .sslist directive
 - description 4-18, 4-66
 - listing control 4-17, 4-36
 - use in macros 5-19
- .ssnolist directive
 - description 4-18, 4-66
 - listing control 4-17, 4-36
 - use in macros 5-19
- stack linker option 7-16, 7-84
- .stack section 7-16, 7-84
- __STACK_SIZE 7-16, 7-57
- .stag COFF debugging directive B-3
- stag structure tag 4-22, 4-70
- START() linker operator 7-59
- STAT symbol reference type 9-5
- .state16 directive 4-10, 4-67
- .state32 directive 4-10, 4-68

- static symbols, creating with `-h` option 7-10
- static variables A-12
 - definition E-7
- status registers 3-24
- storage classes A-15
 - definition E-8
- `.string` directive
 - description 4-13, 4-69
 - limiting listing with the `.option` directive* 4-60
 - limiting listing with the `.option` directive 4-17
- string functions (substitution symbols)
 - `$$firstch` 5-8
 - `$$iscons` 5-8
 - `$$isdefd` 5-8
 - `$$ismember` 5-8
 - `$$isname` 5-8
 - `$$isreg` 5-8
 - `$$lastch` 5-8
 - `$$symcmp` 5-8
 - `$$symlen` 5-8
- string table
 - definition E-8
 - structure A-14
- strip utility
 - invoking 10-17
 - option 10-17
- strip470 utility, invoking 10-17
- stripping
 - line number entries 7-15
 - symbolic information 7-15
- `.struct` directive 4-22, 4-70
- structure
 - definition of E-8
 - stag 4-22, 4-70
- style and symbol conventions iv to viii
- STYP_CLINK flag 4-7
- subsection, defined E-8
- substitution symbols 3-25
 - accessing characters of subscripted 5-11
 - arithmetic operations on 4-21, 5-7
 - as local variables in macros 5-12
 - assigning character strings to 3-25, 4-21
 - built-in functions 5-7 to 5-8
 - defining 5-6
 - description 3-25
 - directives that define 5-6 to 5-7
 - expansion listing 4-18, 4-66
 - finding substrings 5-7

- substitution symbols (continued)
 - forcing substitution 5-9 to 5-10
 - functions and return values 5-7
 - in macros 5-5 to 5-12
 - maximum number per macro 5-5
 - passing commas and semicolons 5-5
 - recursive 5-9
 - subscripted 5-11 to 5-12
 - `.var` directive 5-12
- `.sym` COFF debugging directive B-3
- symbol
 - as labels 3-19
 - assembler-defined 2-17 to 2-18, 3-5
 - assembler usage 3-19 to 3-25
 - assigning SPC to 7-55
 - assigning value to 4-63
 - at link time* 7-54 to 7-61
 - attributes 3-38
 - character strings 3-18
 - cross-reference lister 9-5
 - defined by linker 7-57
 - definition E-8
 - definitions (cross-reference list) 3-37
 - description 3-19 to 3-25
 - external 2-17, 4-44
 - global 7-10
 - in a COFF file 2-17 to 2-18
 - linker-defined 7-57
 - name format A-14
 - number of statements that reference 3-37
 - predefined 3-23
 - reference type 9-5
 - relocatable symbols in expressions 3-28 to 3-30
 - reserved words 7-23
 - setting to a constant value 3-21
 - statement number that defines 3-37
 - storage class A-15
 - table
 - auxiliary entries* A-16
 - contents* A-12
 - creating entries* 2-18
 - definition* E-8
 - entry contents* A-13
 - index* A-10
 - placing unresolved symbols in* 7-18
 - section number* A-16
 - special symbols used in* A-13
 - storage classes* A-15
 - string table structure* A-14
 - stripping entries* 7-15

symbol, table (continued)
 structure and content A-12 to A-16
 symbol name format A-14
 symbol values A-15
 undefining assembler-defined symbols 3-6
 unresolved 7-18
 used as labels 3-19 to 3-20
 value assigned 3-37
 value in symbol table A-15

symbolic constants 3-23
 . symbol 7-55
 \$ 3-23
 coprocessor IDs 3-23
 defining 3-21
 processor symbols 3-24
 register symbols 3-23
 32-BIS registers 3-23
 coprocessor registers 3-23
 status registers 3-24

symbolic debugging B-1 to B-14
 -b linker option 7-8
 COFF directives B-3
 COFF directives syntax B-4
 definition E-8
 directives B-1 to B-14
 disable merge for linker (-b option) 7-8
 DWARF directives B-2
 DWARF directives syntax B-4
 producing error messages in macros 5-17
 put all symbols in symbol table (-as assembler option) 3-5
 stripping symbolic information (-s linker option) 7-15

symbols
 case 3-4
 defined only for C support 7-57

syntax of assignment statements 7-54

.system section 7-11

__SYSTEM_SIZE 7-11, 7-57

system stack, C language 7-16, 7-84

T

t archiver command 6-4
 -t hex conversion utility option 11-5, 11-29
 T operand (.option directive) 4-17
 -t option, name utility 10-16

.tab directive 4-18, 4-72
 table() linker operator 7-70
 used to manage object components 7-71
 .tag directive 4-22, 4-70
 target memory definition E-8
 target width 11-9
 Tektronix object format 11-1, 11-30
 .text directive 2-4, 4-7, 4-73
 .text section 4-7, 4-73, A-3
 definition E-8
 .text symbol defined by linker 7-57
 TI-Tagged object format 11-1, 11-29
 .title directive 4-18, 4-74
 .TMS470... processor symbols 3-24
 TMS470R1x device 1-4
 trampolines, generating 7-16
 --trampolines linker option 7-16

U

u archiver command 6-4
 -u option
 assembler 3-6
 linker 7-18
 name utility 10-16

unconfigured memory 7-26
 definition E-8
 overlying 7-51

undefined output section message (-w linker option) 7-19

underflow (in expression) 3-27

UNDF symbol reference type 9-5

uninitialized sections 2-4 to 2-5, 7-62
 .bss section 2-5, 4-28
 definition E-8
 initialization of 7-65
 specifying a run address 7-45
 .usect section 2-5, 4-76

UNION directive 7-48 to 7-50
 definition E-9

UNION statement, memory overlay example 7-68

unresolved symbol 7-18

.usect directive 2-4, 4-7, 4-76

.utag COFF debugging directive B-3

V

- v archiver option 6-5
- .var directive 4-78, 5-12
 - listing control 4-17, 4-36
- variables
 - autoinitialization at run time 7-84
 - substitution symbols used as 5-12

W

- W MEMORY attribute 7-28
- W operand (.option directive) 4-18
- w option, linker 7-19
- well-defined expressions 3-28
 - definition E-9
- .width directive
 - description 4-18, 4-51
 - listing control 4-17
- .wmsg directive
 - description 4-23, 4-37
 - listing control 4-17, 4-36
 - use in macros 5-17
- word, definition of E-9

- word alignment 4-24
- .word directive
 - description 4-13, 4-49
 - limiting listing with the .option directive 4-17 to 4-18, 4-60
- writeback to register operand suffix (!) 3-13

X

- x archiver command 6-5
- X MEMORY attribute 7-28
- X operand (.option directive) 4-18
- x option
 - hex conversion utility 11-5, 11-30
 - linker 7-19
 - object file display 10-2
- XML example using object file display utility 10-9
- XML tags from object file display utility 10-3 to 10-8
- xref470 command 9-3

Z

- zero hex conversion utility option 11-4, 11-24