TEXAS INSTRUMENTS

TMS320C54x Assembly Language Tools

User's Guide

1997 Digital Signal Processing Products

Texas Instruments

Printed in U.S.A., January 1997
SDS

SPRU102B

User's Guide

TMS320C54x Assembly Language Tools

1997

TMS320C54x Assembly Language Tools User's Guide

Literature Number: SPRU102B
January 1997

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Preface

Read This First

About This Manual

The *TMS320C54x 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

Before you can use this book, you should read the *TMS320C54x Code Generation Tools Getting Started* to install the assembly language tools.

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 TMS320C54x DSPs. This book is divided into four parts:

- **Introductory information** gives you an overview of the assembly language development tools and also discusses common object file format (COFF), which helps you to use the TMS320C54x tools more efficiently. *Read Chapter 2, Introduction to Common Object File Format, before using the assembler and linker.*
- **Assembler description** contains detailed information about using the assembler. This section explains how to invoke the assembler and discusses source statement format, valid constants and expressions, assembler output, and assembler directives. It also summarizes the TMS320C54x instruction set alphabetically and describes macro elements.
- **Additional assembly language tools** describes in detail each of the tools provided with the assembler to help you create assembly language source files. For example, Chapter 9 explains how to invoke the linker, how the linker operates, and how to use linker directives. Chapter 12 explains how to use the hex conversion utility.

- **Reference material** provides supplementary information. This section contains technical data about the internal format and structure of COFF object files. It discusses symbolic debugging directives that the TMS320C54x C compiler uses. Finally, it includes hex conversion utility examples, assembler and linker error messages, and a glossary.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays appear in a `special typeface`. 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:

```
2 0001   2f              x       .byte   47
3 0002   32              z       .byte   50
4 0003                           .text
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font 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. Here is an example of command line syntax:

  abs500 *filename*

  abs500 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; you don't enter the brackets themselves. This is an example of a command that has an optional parameter:

  hex500 [*–options*] *filename*

  The **hex500** command has two parameters. The first parameter, *–options*, is optional. Since *options* is plural, you may select several options. The second parameter, *filename*, is required.

- 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, then it will be 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, should begin in column 1.

  symbol **.usect** **"***section name***",** *size in words* [**,** *blocking flag*] [**,** *alignment flag*]

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

- 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 $value_1$ [**,** ... **,** $value_n$]

Note that **.byte** does not begin in column 1.

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

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

Symbol	Definition	Symbol	Definition
AR0–AR7	Auxiliary Registers 0 through 7	PC	Program counter register
B,b	Suffix — binary integer	Q,q	Suffix — octal integer
H,h	Suffix — hexadecimal integer	SP	Stack pointer register
LSB	Least significant bit	ST	Status register
MSB	Most significant bit		

- 'C54x is used throughout this manual to collectively refer to the TMS320C541, TMS320C542, TMS320C543, TMS320C544, TMS320C545, TMS320C546, TMS320C547, TMS320C548, TMS320C549, and TMS320C545L devices.

Related Documentation From Texas Instruments

The following books describe the TMS320C54x 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.

TMS320C54x DSP Reference Set is composed of four volumes that can be ordered as a set with literature number SPRU210. To order an individual book, use the document-specific literature number:

- ***TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals*** (literature number SPRU131) describes the TMS320C54x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, data and program addressing, the instruction pipeline, DMA, and on-chip peripherals. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

- ***TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set*** (literature number SPRU172) describes the TMS320C54x digital signal processor mnemonic instructions individually. Also includes a summary of instruction set classes and cycles.

- ***TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set*** (literature number SPRU179) describes the TMS320C54x digital signal processor algebraic instructions individually. Also includes a summary of instruction set classes and cycles.

- ***TMS320C54x DSP Reference Set, Volume 4: Applications Guide*** (literature number SPRU173) describes software and hardware applications for the TMS320C54x digital signal processor. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TMS320 DSP Development Support Reference Guide (literature number SPRU011) describes the TMS320 family of digital signal processors and the tools that support these devices. Included are code-generation tools (compilers, assemblers, linkers, etc.) and system integration and debug tools (simulators, emulators, evaluation modules, etc.). Also covered are available documentation, seminars, the university program, and factory repair and exchange.

TMS320C54x Optimizing C Compiler User's Guide (literature number SPRU103) describes the 'C54x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C54x generation of devices.

TMS320C5xx C Source Debugger User's Guide (literature number SPRU099) tells you how to invoke the 'C54x emulator, EVM, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C54x Code Generation Tools Getting Started Guide (literature number SPRU147) describes how to install the TMS320C54x assembly language tools and the C compiler for the 'C54x devices. The installation for MS-DOS™, OS/2™, SunOS™, Solaris™, and HP-UX™ 9.0x systems is covered.

TMS320 Third-Party Support Reference Guide (literature number SPRU052) alphabetically lists over 100 third parties that provide various products that serve the family of '320 digital signal processors. A myriad of products and applications are offered—software and hardware development tools, speech recognition, image processing, noise cancellation, modems, etc.

Digital Signal Processing Applications with the TMS320 Family, Volumes 1, 2, and 3 (literature numbers SPRA012, SPRA016, SPRA017) Volumes 1 and 2 cover applications using the 'C10 and 'C20 families of fixed-point processors. Volume 3 documents applications using both fixed-point processors as well as the 'C30 floating-point processor.

Trademarks

If You Need Assistance. . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/mirrors/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68		
English	+33 1 30 70 11 65		
Francais	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10	

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: comments@books.sc.ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C54x DSPs are supported by the following assembly language tools:

- Assembler
- Archiver
- Linker
- Absolute lister
- Cross-reference utility
- Hex conversion utility
- Mnemonic-to-algebraic translator 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; however, the compiler and debugger are not shipped with the assembly language tools. For detailed information on the compiler and debugger and for complete descriptions of the TMS320C54x devices, refer to the books listed in *Related Documentation From Texas Instruments* on page vi.

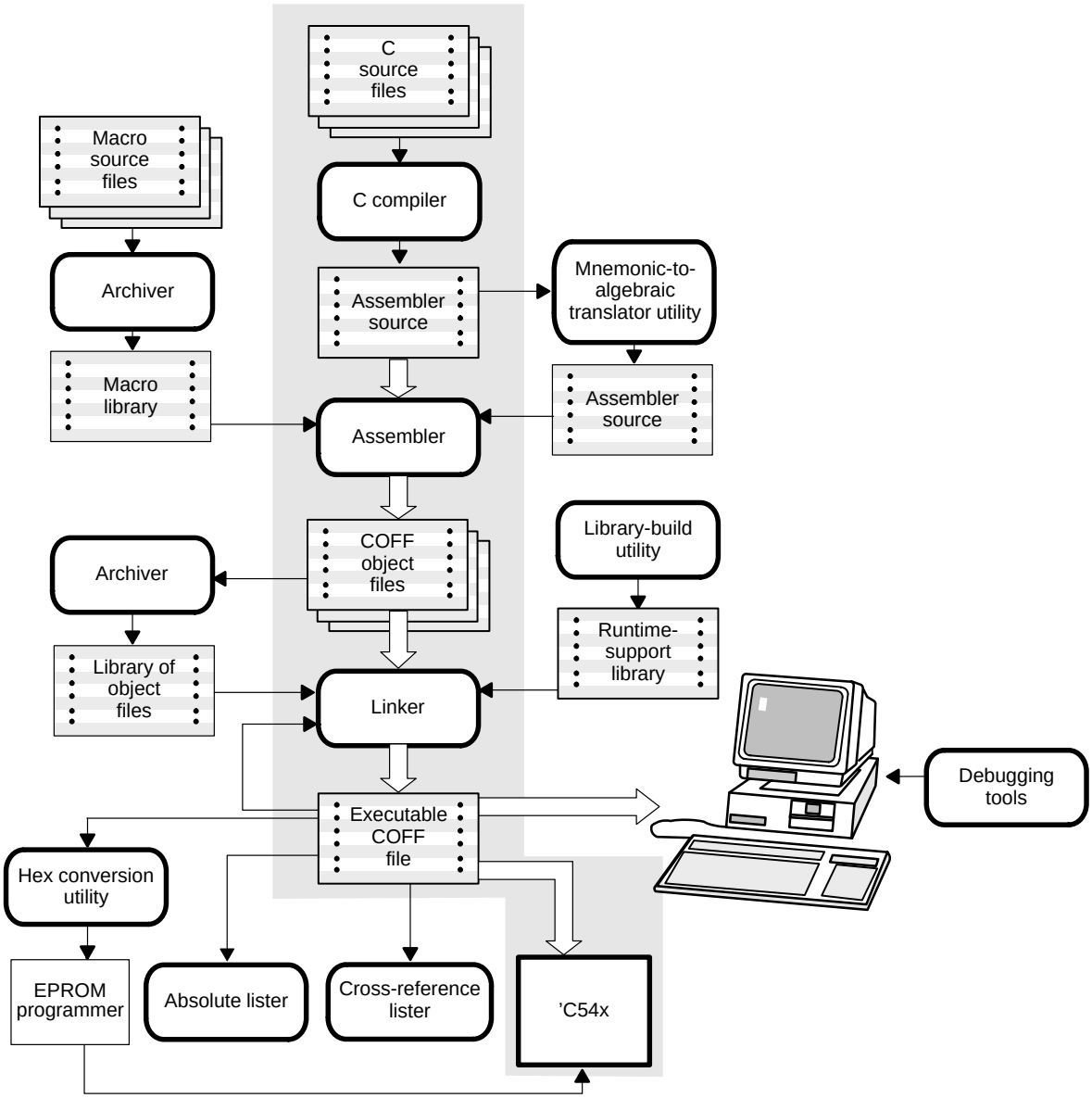
The assembly language tools create and use object files in common object file format (COFF) to facilitate modular programming. Object files contain separate blocks (called sections) of code and data that you can load into ’C54x memory spaces. You can program the ’C54x more efficiently if you have a basic understanding of COFF. Chapter 2, *Introduction to Common Object File Format*, discusses this object format in detail.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 illustrates the ’C54x software development flow. The shaded portion of the figure highlights the most common path of software development; the other portions are optional.

Figure 1–1. TMS320C54x Software Development Flow



1.2 Tools Descriptions

- The **C compiler** translates C source code into 'C54x assembly language source code. The compiler package includes the **library-build utility**, with which you can build your own runtime libraries. The C compiler is not shipped with the assembly language tools package.
- 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.
- The **linker** combines relocatable COFF object files (created by the assembler) into a single executable COFF object module. As it creates the executable module, it adjusts references to symbols and resolves external references. 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.
- The **archiver** collects a group of 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 **mnemonic-to-algebraic translator utility** converts an assembly language source file containing mnemonic instructions to an assembly language source file containing algebraic instructions.
- The **library-build utility** builds your own customized, C, runtime-support library. Standard runtime-support library functions are provided as source code in rts.src and as object code in rts.lib.
- The TMS320C54x DSP accepts COFF files as input, but most EPROM programmers do not. The **hex conversion utility** converts a COFF object file into TI-tagged, Intel, Motorola, or Tektronix object format. The converted file can be downloaded to an EPROM programmer.
- The **absolute lister** accepts linked object files as input and creates .abs files as output. You assemble .abs files 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.

- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files.

The purpose of this development process is to produce a module that can be executed in a ’C54x target system. You can use one of several debugging tools to refine and correct your code. Available products include:

- An instruction-accurate software simulator
- An extended development system (XDS510™) emulator
- An evaluation module (EVM)

For information about these debugging tools, see the *TMS320C54x C Source Debugger User’s Guide*.

Chapter 2

Introduction to Common Object File Format

The assembler and linker create object files that can be executed by a TMS320C54x 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 provides an overview of COFF sections. For additional information, see Appendix A, *Common Object File Format*, which explains the COFF structure.

Topic **Page**

2.1 COFF File Types

The following types of COFF files exist:

- COFF0
- COFF1
- COFF2

Each COFF file type has a different header format. The data portions of the COFF files are identical. For details about the COFF file structure, see Appendix A, *Common Object File Format*.

The 'C54x assembler and C compiler create COFF2 files. The linker can read and write all types of COFF files. By default, the linker creates COFF2 files. Use the –v linker option to specify a different format. The linker supports COFF0 and COFF1 files for older versions of the assembler and C compiler only.

2.2 Sections

The object file is grouped into sections. Sections are blocks of code or data that 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:

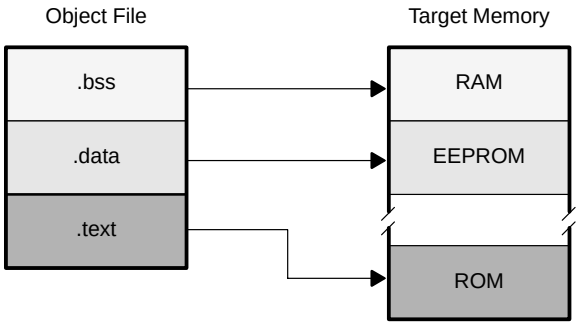
Initialized sections	contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.
Uninitialized sections	reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created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



2.3 How the Assembler Handles Sections

The assembler identifies the portions of an assembly language program that belong in a 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 subsection 2.3.4, *Subsections*, page 2-8, for more information.

Note: Default Section Directive

If you don't use any of the sections directives, the assembler assembles everything into the .text section.

2.3.1 Uninitialized Sections

Uninitialized sections reserve space in 'C54x 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 runtime 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 directive, the assembler reserves more space in the appropriate section. Each time you invoke the .usect directive, the assembler reserves more space in the specified named section.

The syntax for these directives is:

	.bss *symbol, size in words* [, *blocking flag*] [, *alignment flag*]
symbol	**.usect** "*section name* ", *size in words* [, *blocking flag*] [, *alignment flag*]

symbol points to the first word reserved by this invocation of the .bss or .usect directive. The *symbol* corresponds to the name of the variable that you're reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the .global assembler directive).

size in words is an absolute expression.

- The .bss directive reserves *size* words in the .bss section.
- The .usect directive reserves *size* words in *section name.*

blocking flag is an optional parameter. If you specify a value greater than 0 for this parameter, the assembler associates size words contiguously; the allocated space will not cross a page boundary, unless size is greater than a page, in which case the object will start on a page boundary.

alignment flag is an optional parameter. If you specify a value greater than 0 for this parameter, the section is aligned to a long word boundary.

section name tells the assembler which named section to reserve space in. For more information about named sections, see subsection 2.3.3, *Named Sections*, on page 2-7.

The .text, .data, and .sect directives 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 temporarily from the current section. The .bss and .usect directives can appear anywhere in an initialized section without affecting its contents.

Uninitialized subsections are created with the .usect directive. The assembler treats uninitialized subsections in the same manner as uninitialized sections. See subsection 2.3.4, *Subsections*, on page 2-8 for more information on creating subsections.

2.3.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in 'C54x memory when the program is loaded. Each initialized section is independently relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section-relative references.

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

.text [*value*]
.data [*value*]
.sect "*section name***"** [**,** *value*]

When the assembler encounters one of these directives, it stops assembling into the current section (acting as an implied end-current-section command). It then assembles subsequent code into the designated section until it encounters another .text, .data, or .sect directive. The *value*, if present, specifies the starting value of the section program counter. The starting value of the section program counter can be specified only once; it must be done the first time the directive for that section is encountered. By default, the SPC starts at 0.

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

Initialized subsections are created with the .sect directive. The assembler treats initialized subsections in the same manner as initialized sections. See subsection 2.3.4, *Subsections*, on page 2-8 for more information on creating subsections.

2.3.3 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 don't 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 sections that are used like the .bss section. These sections reserve space in RAM for variables.
- The **.sect** directive creates 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 words* [, *blocking flag*] [, *alignment flag*]
.sect "*section name*"

The *section name* parameter is the name of the section. You can create up to 32 767 separate named sections. A section name can be up to 200 characters. For COFF1 formatted files, only the first 8 characters are significant. For the .sect and .usect directives, a section name can refer to a subsection (see subsection 2.3.4, *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.

2.3.4 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 a subsection name is:

section name:*subsection name*

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

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

You can allocate _func separately or within all the .text sections.

You can create two types of subsections:

- Initialized subsections are created using the .sect directive. See subsection 2.3.2, *Initialized Sections*, on page 2-6.
- Uninitialized subsections are created using the .usect directive. See subsection 2.3.1, *Uninitialized Sections*, on page 2-4.

Subsections are allocated in the same manner as sections. See Section 9.8, *The SECTIONS Directive*, on page 9-30 for more information.

2.3.5 Section Program Counters

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

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

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 2.5, *Relocation*, on page 2-14.

2.3.6 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

```
      2                 ************************************************
      3                 **  Assemble an initialized table into .data. **
      4                 ************************************************
      5  0000                       .data
      6  0000   0011    coeff      .word        011h,022h,033h
         0001   0022
         0002   0033
      7                 ************************************************
      8                 **   Reserve space in .bss for a variable.    **
      9                 ************************************************
     10  0000                       .bss         buffer,10
     11                ************************************************
     12                **              Still in .data.               **
     13                ************************************************
     14  0003   0123    ptr        .word        0123h
     15                ************************************************
     16                **    Assemble code into the .text section.   **
     17                ************************************************
     18  0000                       .text
     19  0000   100f    add:       LD           0Fh,A
     20  0001   f010    aloop:     SUB          #1,A
         0002   0001
     21  0003   f842               BC           aloop,AGEQ
         0004   0001'
     22                ************************************************
     23                **    Another initialized table into .data.   **
     24                ************************************************
     25  0004                       .data
     26  0004   00aa    ivals      .word        0AAh, 0BBh, 0CCh
         0005   00bb
         0006   00cc
     27                ************************************************
     28                ** Define another section for more variables. **
     29                ************************************************
     30  0000          var2       .usect       "newvars", 1
     31  0001          inbuf      .usect       "newvars", 7
     32                ************************************************
     33                **       Assemble more code into .text.       **
     34                ************************************************
     35  0005                       .text
     36  0005   110a    mpy:       LD              0Ah,B
     37  0006   f166    mloop:     MPY             #0Ah,B
         0007   000a
     38  0008   f868               BC              mloop,BNOV
         0009   0006'
     39                ************************************************
     40                **  Define a named section for int. vectors.  **
     41                ************************************************
     42  0000                       .sect           "vectors"
     43  0000   0011               .word           011h, 033h
     44  0001   0033
```

Field 1 Field 2 Field 3 Field 4

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

.text	contains ten 16-bit words of object code.
.data	contains seven words of object code.
vectors	is a named section created with the .sect directive; it contains two words of initialized data.
.bss	reserves 10 words in memory.
newvars	is a named section created with the .usect directive; it reserves eight words in memory.

The second column shows the object code that is assembled into these sections; the first column shows the line numbers of 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 20 21 21 36 37 37 38 38	100f f010 0001 f842 0001’ 110a f166 000a f868 0006’	.text
6 6 6 14 26 26 26	0011 0022 0033 0123 00aa 00bb 00cc	.data
43 44	0011 0033	vectors
10	No data— 10 words reserved	.bss
30 31	No data— eight words reserved	newvars

2.4 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 don't use them, the linker uses the target processor's default allocation algorithm described in Section 9.12, *Default Allocation Algorithm*, on page 9-51. When you *do* use linker directives, you must specify them in a linker command file.

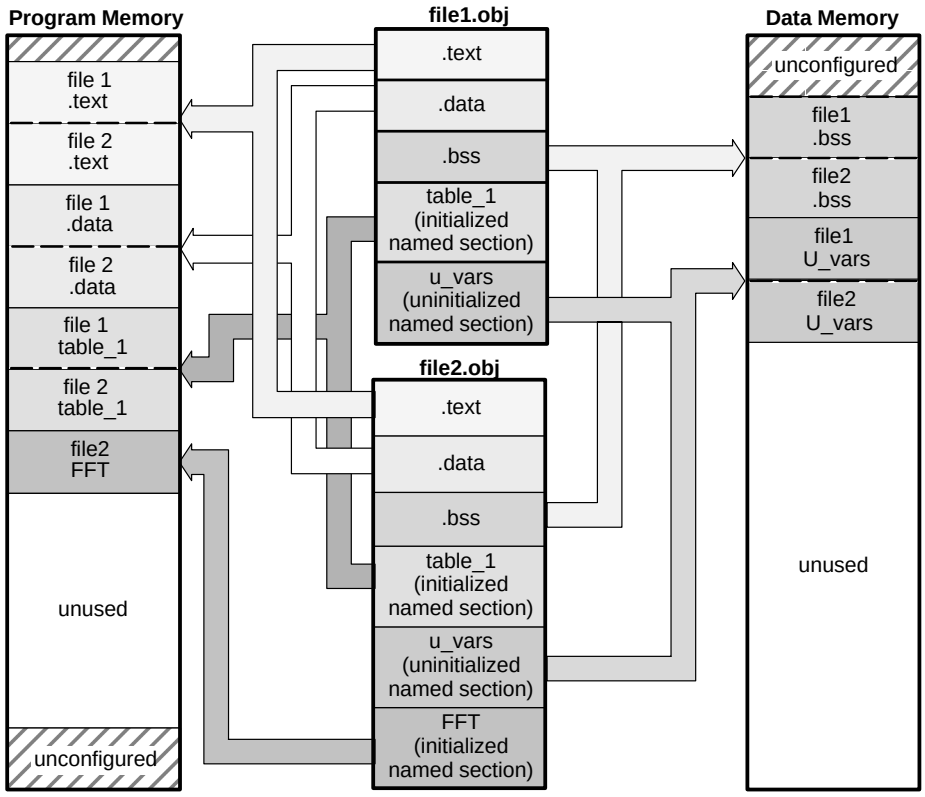
Refer to the following sections for more information about linker command files and linker directives:

Section Number	Section Name	Page
9.5	Linker Command Files	9-21
9.7	The MEMORY Directive	9-26
9.8	The SECTIONS Directive	9-30
9.12	Default Allocation Algorithm	9-51

2.4.1 Default Memory Allocation

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

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 named sections. The executable output module shows the combined sections. The linker combines file1.text with file2.text to form one .text section, then combines the .data sections, then the .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 080h and places the sections one after the other as shown.

2.4.2 Placing Sections in the Memory Map

Figure 2–3 illustrates the linker's default methods 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 Section 9.7, *The MEMORY Directive*, on page 9-26 and Section 9.8, *The SECTIONS Directive*, on page 9-30.

2.5 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 can't 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
- ❏ 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 'C54x that generates relocation entries.

Example 2–2. Code That Generates Relocation Entries

(a) Mnemonic example

```
1        0100   X   .set 0100h
2  000000           .text
3  000000 F073      B   Y      ; Generates a Relocation Entry
   000001 0004'
4  000002 F020      LD #X, A   ; Generates a Relocation Entry
   000003 0000!
5  000004 F7E0   Y:  RESET
```

(b) Algebraic example

```
1        0100   X   .set 0100h
2  000000         .text
3  000000 F073    goto   Y      ; Generates a Relocation Entry
   000001 0004'
4  000002 F020    A = #X        ; Generates a Relocation Entry
   000003 0000!
5  000004 F7E0    Y: reset
```

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 3 (relative to address 0 in the .text section). The assembler generates two relocation entries, one for X and one for Y. The reference to X is an external reference (indicated by the **!** character in the listing). The reference to Y is to an internally defined relocatable symbol (indicated by the **'** character in the listing).

After the code is linked, suppose that X is relocated to address 7100h. Suppose also that the .text section is relocated to begin at address 7200h; Y now has a relocated value of 7204h. The linker uses the two relocation entries to patch the two references in the object code:

```
f073   B  Y       becomes   f073
0004'                       7204'
f020   LD #X,A    becomes   f020
0000!                       7100!
```

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.

2.6 Runtime 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 ROM, but it would run faster in RAM.

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 will place the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this does *not* happen automatically simply because you specify a separate run address. For an example that illustrates how to move a block of code at runtime, see Example 9–6 on page 9-41.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and will load and run 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 runtime relocation, see Section 9.9, *Specifying a Section's Runtime Address*, on page 9-39.

2.7 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 below.

- The TMS320C54x debugging tools, including the software simulator, XDS51x emulator, and software development system, 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 (hex500, 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.

2.8 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.

2.8.1 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** directives to identify symbols as external:

.def	Defined in the current module and used in another module
.ref	Referenced in the current module, but defined in another module
.global	May be either of the above

The following code segment illustrates these definitions.

```
x:  ADD     #56h, A     ; Define x
    B       y           ; Reference y
   .def     x           ; DEF of x
   .ref     y           ; REF of y
```

The .def definition of x says that it is an external symbol defined in this module and that other modules can reference x. The .ref definition of y says that it is an undefined symbol that is defined in another module.

The 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 defines unresolved references to x from 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.

2.8.2 The Symbol Table

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

The assembler does not usually create symbol table entries for any symbols other than those described above, because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with .global. 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 –s option.

Chapter 3

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
Assembly language instructions	described in Chapters 5 and 6
Macro directives	described in Chapter 7

Topic **Page**

3.1 Assembler Overview

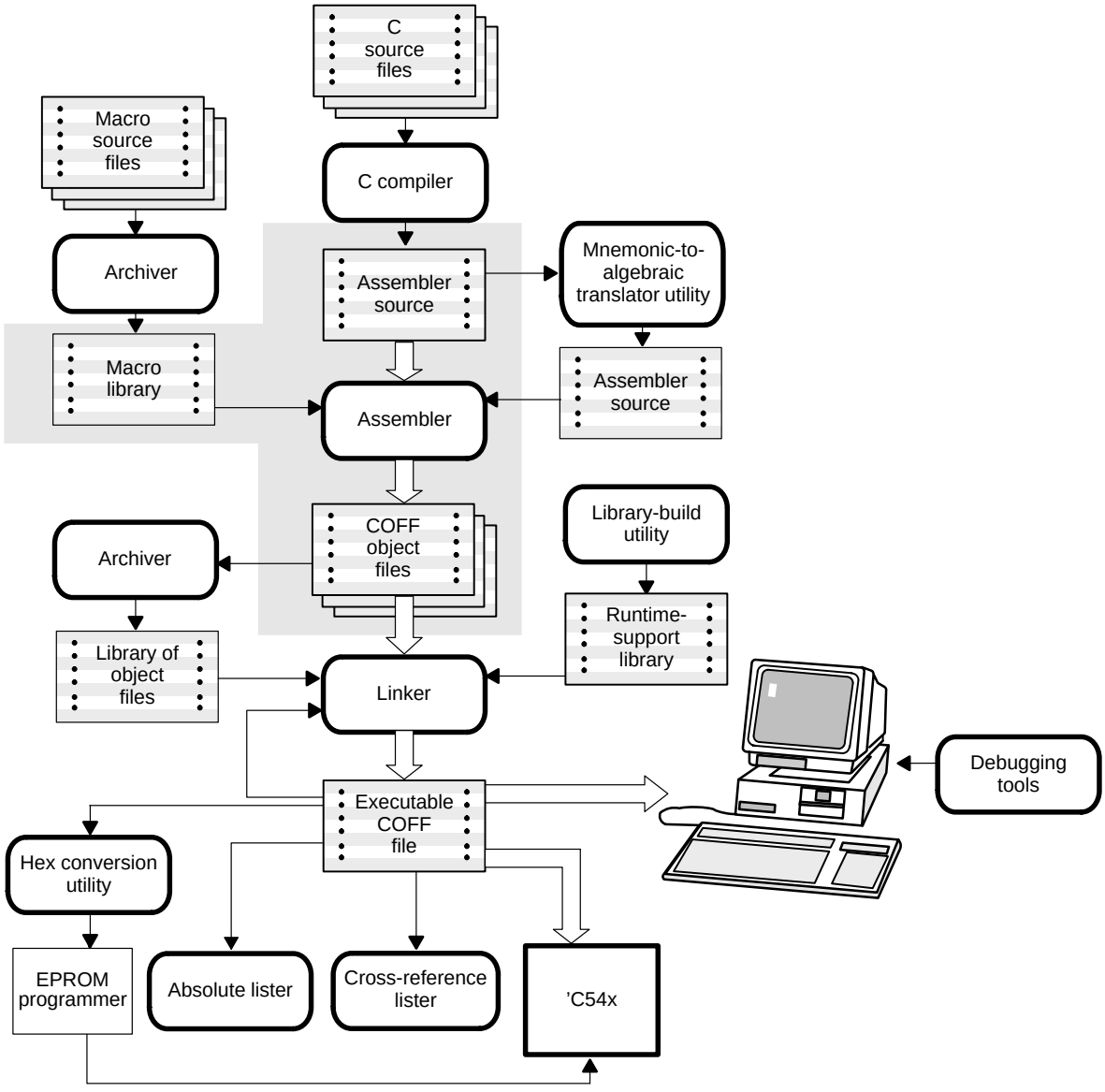
This two-pass assembler does the following:

- 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)
- Assembles conditional blocks
- Supports macros, allowing you to define macros inline or in a library

3.2 Assembler Development Flow

Figure 3–1 illustrates the assembler's role in the assembly language development flow. The assembler accepts assembly language source files as input, whether created by the assembler itself or by the C compiler.

Figure 3–1. Assembler Development Flow



3.3 Invoking the Assembler

To invoke the assembler, enter the following:

asm500 [*input file* [*object file* [*listing file*]]] [*–options*]

asm500 is the command that invokes the assembler.

input file names the assembly language source file. If you do not supply an extension, the assembler uses the default extension *.asm*. If you do not supply an input filename, the assembler prompts you for one.

The source file can contain mnemonic or algebraic instructions, but not both. The default instruction set is mnemonic. To specify the algebraic instruction set, use the –mg option.

object file names the object file that the assembler creates. If you do not supply an extension, the assembler uses *.obj* as a default. If you do not supply an object file, the assembler creates a file that uses the input filename with the *.obj* extension.

listing file names the optional listing file that the assembler can create. If you do not supply a listing filename, *the assembler does not create one* unless you use the –l (lowercase L) option. In this case, the assembler uses the input filename. If you do not supply an extension, the assembler uses *.lst* as a default.

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. Single-letter options without parameters can be combined: for example, –lc is equivalent to –l –c. Options that have parameters, such as –i, must be specified separately.

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

–c makes case insignificant in the assembly language files. For example, –c will make the symbols ABC and abc equivalent. *If you do not use this option, case is significant* (default).

–d **–d***name* [=*value*] sets the *name* symbol. This is equivalent to inserting *name* **.set** [*value*] at the beginning of the assembly file. If *value* is omitted, the symbol is set to 1. For more information, see subsection 3.8.3, *Defining Symbolic Constants (–d Option),* on page 3-17.

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

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

–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*. You can specify up to 10 directories in this manner; each pathname must be preceded by the –i option. For more information, see subsection 3.4.1, *–i Assembler Option*, on page 3-6.

–l (lowercase L) produces a listing file.

–mg specifies that the source file contains algebraic instructions.

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

–s 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 –s, symbols defined as labels or as assembly-time constants are also placed in the table.

–x produces a cross-reference table and appends it to the end of the listing file; also adds cross-reference information to the object file for use by the cross-reference utility. If you do not request a listing file, the assembler creates one anyway.

3.4 Naming Alternate Files and 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. 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 order:

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 environment variable A_DIR

You can augment the assembler's directory search algorithm by using the –i assembler option or the A_DIR environment variable.

3.4.1 –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:

asm500 –i*pathname source filename*

You can use up to 10 –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 doesn't 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 that the file is stored in the following directory:

DOS or OS/2™ c:\320tools\files\copy.asm

UNIX™ /320tools/files/copy.asm

Operating System	Enter
DOS or OS/2	`asm500 –ic:\320tools\files source.asm`
UNIX	`asm500 –i/320tools/files source.asm`

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 A_DIR Environment Variable

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

Operating System	Enter
DOS or OS/2	`set A_DIR= pathname;another pathname ...`
UNIX	`setenv A_DIR "pathname;another pathname ..."`

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 doesn't find the file in the directory that contains the current source file or in directories named by –i, 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 that the files are stored in the following directories:

DOS or OS/2 c:\320tools\files\copy1.asm
c:\dsys\copy2.asm

UNIX /320tools/files/copy1.asm
/dsys/copy2.asm

You could set up the search path with the commands shown in the following table:

Operating System	Enter
DOS or OS/2	`set A_DIR=c:\dsys` `asm500 –ic:\320tools\files source.asm`
UNIX	`setenv A_DIR "/dsys"` `asm500 –i/320tools/files source.asm`

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
DOS or OS/2	`set A_DIR=`
UNIX	`unsetenv A_DIR`

3.5 Source Statement Format

TMS320C54x assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. Source statement lines can be as long as the source file format allows, but the assembler reads up to 200 characters per line. If a statement contains more than 200 characters, the assembler truncates the line and issues a warning.

Example 3–1. Example Source Statements

(a) Mnemonic instructions

```
SYM1          .set      2             ; Symbol SYM1 = 2.
Begin:         LD       #SYM1, AR1    ; Load AR1 with 2.
              .word     016h          ; Initialize word (016h)
```

(b) Algebraic instructions

```
SYM1          .set      2             ; Symbol SYM1 = 2.
Begin:         AR1 = #SYM1            ; Load AR1 with 2.
              .word     016h          ; Initialize word (016h)
```

3.5.1 Source Statement Syntax

A source statement can contain four ordered fields. The general syntax for source statements is as follows:

Mnemonic syntax:

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

Algebraic syntax:

[*label*] [:] *instruction* [;*comment*]

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 characters are equivalent to blanks.
- 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.

3.5.2 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 32 alphanumeric characters (A–Z, a–z, 0–9, _, and $). Labels are case sensitive, 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 don't 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 (the label points to the statement it's associated with). If, for example, 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    003F         ; Assume some other code was assembled.
10    0040  000A   Start:  .word 0Ah,3,7
      0041  0003
      0042  0007
```

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   0050            Here:
4   0050   0003        .word 3
```

3.5.3 Mnemonic Field

The mnemonic field follows the label field. The mnemonic field must not start in column 1; if it does, it will be interpreted as a label. The mnemonic field can contain one of the following opcodes:

- Machine-instruction mnemonic (such as ABS, MPYU, STH)
- Assembler directive (such as .data, .list, .set)
- Macro directive (such as .macro, .var, .mexit)
- Macro call

3.5.4 Operand Field

The operand field is a list of operands that follow the mnemonic field. An operand can be a constant (see Section 3.6, *Constants*, on page 3-13), a symbol (see Section 3.8, *Symbols*, on page 3-16), or a combination of constants and symbols in an expression (see Section 3.9, *Expressions*, on page 3-23). You must separate operands with commas.

❑ **Operand Prefixes for Instructions**

The assembler allows you to specify that a constant, symbol, or expression should be used as an address, an immediate value, or an indirect value. 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 or an address; the assembler treats the address as a value instead of using the contents of the address. This is an example of an instruction that uses an operand with the # prefix:

  ```
  Label:  ADD #123,B
  ```

  The operand #123 is an immediate value. The assembler adds 123 (decimal) to the contents of accumulator B.

- *** prefix — the operand is an indirect address.** If you use the ***** sign as a prefix, the assembler treats the operand as an indirect address; that is, it uses the contents of the operand as an address. This is an example of an instruction that uses an operand with the * prefix:

  ```
  Label:  LD *AR4,A
  ```

  The operand *AR4 specifies an indirect address. The assembler goes to the address specified by the contents of register AR4 and then moves the contents of that location to accumulator A.

❑ **Immediate Value for Directives**

The immediate value mode is primarily used with instructions. In some cases, it can also be used with the operands of directives.

It is not usually necessary to use the immediate value mode for directives. Compare the following statements:

```
ADD #10, A
.byte 10
```

In the first statement, the immediate value mode is necessary to tell the assembler to add the value 10 to accumulator A. In the second statement, however, immediate value is not used; the assembler expects the operand to be a value and initializes a byte with the value 10.

3.5.5 Instruction Field

The instruction field is a combination of the mnemonic and operand fields used in mnemonic syntax. In algebraic syntax, you usually do not have a mnemonic followed by operands. Rather, the operands are part of the overall statement. The following items describe how to use the instruction field for mnemonic and algebraic syntax:

- Generally, operands are not separated by commas. Some algebraic instructions, however, consist of a mnemonic and operands. For algebraic statements of this type, commas are used to separate operands. For example, lms(Xmem, Ymem).
- Expressions that have more than one term that is used as a single operand must be delimited with parentheses. This rule does not apply to statements using a function call format, since they are already set off with parentheses. For example, A = B & #(1 << sym) << 5. The expression 1 << sym is used as a single operand and is therefore set off with parentheses.
- All register names are reserved.
- For algebraic instructions that consist of a mnemonic and operands, the mnemonic word is reserved.

3.5.6 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 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
- Floating-point

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.

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 zero fills the unspecified bits. 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 6 octal digits (0 through 7) prefixed with a **0** (zero) or suffixed with **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 $8\ 000_{16}$
226q	Constant equal to 150_{10} or 96_{16}
010	Constant equal to 8_{10} or 8_{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 $8\ 000_{16}$
25	Constant equal to 25_{10} or 19_{16}

3.6.4 Hexadecimal Integers

A hexadecimal integer constant is a string of up to four 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 one or two characters 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. If only one character is specified, the assembler right-justifies the bits. These are examples of valid character constants:

'a'	Represented internally as 61_{16}
'C'	Represented internally as 43_{16}
'''D'	Represented internally as 2 744_{16}

Note the difference between character constants and character strings (Section 3.7, *Character Strings*, on page 3-15, discusses character strings). A character constant represents a single integer value; a string is a list of characters.

3.6.6 Assembly-Time Constants

If you use the .set directive 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
           LD       #shift3,A
```

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   AR1
           MVMM AuxR1, SP
```

3.6.7 Floating-Point Constants

A floating-point constant is a string of decimal digits, followed by an optional decimal point, fractional portion, and exponent portion. The syntax for a floating-point number is:

[+|–] [*nnn*] **.** [*nnn* [**E|e** [**+|–**] *nnn*]]

Replace *nnn* with a string of decimal digits. You can precede *nnn* with a + or a –. You must specify a decimal point. For example, 3.e5 is valid, but 3e5 is not valid. The exponent indicates a power of 10. These are examples of valid character constants:

3.0
3.14
.3
–0.314e13
+314.59e–2

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 –c assembler option. A symbol is valid only during the assembly in which it is defined, unless you use the .global directive to declare it as an external symbol.

3.8.1 Labels

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

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

```
          .global   label1

label2    NOP
          ADD       label1,B
          B         label2
```

3.8.2 Symbolic Constants

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

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

item   .struct                 ;item structure definition
       .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 the next section.

3.8.3 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:

asm500 –d*name*=[*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.

Within assembler source, you may test the symbol with the following directives:

Type of Test	Directive Usage
Existence	`.if $isdefed("name")`
Nonexistence	`.if $isdefed("name") = 0`
Equal to value	`.if name = value`
Not equal to value	`.if name != value`

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

3.8.4 Predefined Symbolic Constants

The assembler has several predefined symbols, including the following:

- **$**, the dollar sign character, represents the current value of the section program counter (SPC).
- **Register symbols**, including AR0 – AR7
- **Symbols** that are defined by using the .mmregs directive (see Table 4–9 on page 4-62 for a list of registers)

3.8.5 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  ”errct”,     AR2   ;register 2
   .asg  ”*+”,        INC   ;indirect auto-increment
   .asg  ”*–”,        DEC   ;indirect auto-decrement
```

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:

```
add2   .macro    ADDRA,ADDRB  ;add2 macro definition

       LD ADDRA, A
       ADD ADDRB, A
       STL A, ADDRB

       .endm

*add2 invocation
       add2 LOC1, LOC2        ;add ”LOC1” argument to a
                              ;second argument ”LOC2”.
       LD LOC1, A
       ADD LOC2, A
       STL A, LOC2
```

For more information about macros, see Chapter 7, *Macro Language*.

3.8.6 Local Labels

Local labels are special labels whose scope and effect are temporary. 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 or automatically generated.

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.

A local label takes one of the following forms:

- $*n*
- *name*?

3.8.6.1 $n Form

In the $n form, *n* is a decimal digit in the range 0–9. For example, $4 and $1.

The label is made unique by undefining the label before using it again. Undefine a label in one of these ways:

- Use the .newblock directive
- Change sections using a .sect, .text, or .data directive
- Enter an include file using the .include or .copy directive
- Leave an include file using the .include or .copy directive

Example 3–2 demonstrates the $*n* form.

Example 3–2. $n Local Labels

(a) Code that uses a local label legally

```
Label1:   LD ADDRA, A    ; Load Address A to Accumulator A.
          SUB ADDRB, A   ; Subtract Address B.
          BC $1, ALT     ; If less than zero, branch to $1;
          LD ADDRB, A    ; otherwise, load ADDRB to A
          B $2           ; and branch to $2.
$1        LD ADDRA, A    ; $1: load ADDRA to Accumulator A.
$2        ADD ADDRC, A   ; $2: add ADDRC.
          .newblock      ; Undefine $1 so it can be used
                         ; again.
          BC $1, ALT     ; If less than zero, branch to $1.
          STL A, ADDRC   ; Store ACC low in ADDRC.
$1        NOP
```

(b) Code that uses a local label illegally

```
Label1:   LD ADDRA, A    ; Load Address A to Accumulator A.
          SUB ADDRB, A   ; Subtract Address B.
          BC $1, ALT     ; If less than zero, branch to $1;
          LD ADDRB, A    ; otherwise, load ADDRB To A
          B $2           ; and Branch to $2.
$1        LD ADDRA, A    ; $1: Load ADDRA To Accumulator A.
$2        ADD ADDRC, A   ; $2: Add ADDRC.
          BC $1, ALT     ; If less than 0, branch to $1.
          STL A, ADDRC   ; Store Acc low in ADDRC.
$1        NOP            ; WRONG: $1 is multiply defined.
```

3.8.6.2 name? Form

In the *name*? form, *name* is any legal symbol name. For example, here?.

The assembler replaces the question mark with a unique number. When the macro 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 macro definition. You can see the label with its unique number in the cross-listing file. You cannot declare this label as global.

The maximum label length is shortened to allow for the unique suffix. 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.

Example 3–3 demonstrates the *name*? form.

Example 3–3. name? Local Labels

(a) Source code

```
;*******************************************************
;  First definition of local label 'mylab'
;*******************************************************
'
          nop
mylab?    nop
          b mylab?
;*******************************************************
;  Include file has second definition of 'mylab'
;*******************************************************
'
          .copy "a.inc"
;*******************************************************
;Third definition of 'mylab', reset upon exit from include
;*******************************************************
'
mylab?    nop
          b mylab?
```

Example 3–3.name? Local Labels (Continued)

(a) Source code (continued)

```
;*********************************************************
;  Fourth definition of 'mylab' in macro, macros use
;  different namespace to avoid conflicts
;*********************************************************
mymac     .macro
mylab?    nop
          b mylab?
          .endm
;*********************************************************
;  Macro invocation
;*********************************************************
          mymac
;*********************************************************
;  Reference to third definition of 'mylab', note that
;  definition is not reset by macro invocation nor
;  conflicts with same name defined in macro
;*********************************************************
          b mylab?;
*********************************************************
;  Changing section, allowing fifth definition of 'mylab'
;*********************************************************
          .sect "Secto_One"
          nop
mylab?    .word 0
          nop
          nop
          b mylab?
;*********************************************************
;.newblock directive, allowing sixth definition of 'mylab'
;*********************************************************
          .newblock
mylab?    .word 0
          nop
          nop
          b mylab?
```

(b) Resulting assembly listing file

```
    1           ;*********************************************************
    2           ;  First definition of local label 'mylab'
    3           ;*********************************************************
    4 0000 5500         nop
    5 0001 5500  mylab? nop
    6 0002 ff80         b mylab?
    7           ;*********************************************************
    8           ;  Include file has second definition of 'mylab'
    9           ;*********************************************************
   10                   .copy "a.inc"
A   1 0004 5500  mylab? nop
```

Example 3–3.name? Local Labels (Continued)

(b) Resulting assembly listing file (continued)

```
A  2  0005 ff80        b mylab?
      0006 0004'
   11            ;*********************************************************
   12            ;Third definition of 'mylab', reset upon exit from include
   13            ;*********************************************************
   14  0004 5500 mylab? nop
   15  0005 ff80        b mylab?
      0009 0007'
   16            ;*********************************************************
   17            ;  Fourth definition of 'mylab' in macro, macros use
   18            ;  different namespace to avoid conflicts
   19            ;*********************************************************
   20            mymac  .macro
   21            mylab? nop
   22                   b mylab?
   23                   .endm
   24            ;*********************************************************
   25            ;  Macro invocation
   26            ;*********************************************************
   27  000a             mymac
1     000a 5500 mylab? nop
1     000b ff80        b mylab?
      000c 000a'
   28            ;*********************************************************
   29            ;  Reference to third definition of 'mylab', note that
   30            ;  definition is not reset by macro invocation nor
   31            ;  conflicts with same name defined in macro
   32            ;*********************************************************
   33  000d ff80        b mylab?
      000e 0007'
   34            ;*********************************************************
   35            ;  Changing section, allowing fifth definition of 'mylab'
   36            ;*********************************************************
   37  0000             .sect "Secto_One"
   38  0000 5500        nop
   39  0001 0000 mylab? .word 0
   40  0002 5500        nop
   41  0003 5500        nop
   42  0004 ff80        b mylab?
      0005 0001+
   43            ;*********************************************************
   44            ;.newblock directive, allowing sixth definition of 'mylab'
   45            ;*********************************************************
   46                   .newblock
   47  0006 000  mylab? .word 0
   48  0007 5500        nop
   49  0008 5500        nop
   50  0009 ff80        b mylab?
      000a 0006+
```

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 –32 768 to 32 767. Three main factors influence the order of expression evaluation:

Parentheses

Expressions that are enclosed in parentheses are always evaluated first.

8 / (4 / 2) = 4, but 8 / 4 / 2 = 1

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

Precedence groups

The ’C54x assembler uses the same order of precedence as the C language does as summarized in Table 3–1. This differs from the order of precedence of the ’C54x assembler. When parentheses do not determine the order of expression evaluation, the highest precedence operation is evaluated first.

8 + 4 / 2 = 10 (4 / 2 is evaluated first)

Left-to-right evaluation

When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated as happens in the C language.

8 / 4*2 = 4 , but 8 / (4*2) = 1

3.9.1 Operators

Table 3–1 lists the operators that can be used in expressions.

Note: Differences in Precedence From Other TMS320 Assemblers

Some other TMS320 processors use a different order of precedence than the TMS320C54x, and occasionally different results may be produced from the same source code for this reason. The ’C54x uses the same order of precedence as the C language.

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

Symbols	Operators	Evaluation
+ – ~	Unary plus, minus, 1s complement	Right to left
* / %	Multiplication, division, modulo	Left to right
+ –	Addition, subtraction	Left to right
<< >>	Left shift, right shift	Left to right
< <= > >=	Less than, LT or equal, greater than, GT or equal	Left to right
!=, =[=]	Not equal to, equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	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 Value Truncated warning 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.

Example 3–4. Well-Defined Expressions

(a) Valid well-defined expressions

```
label1   .word  0
         .word  1
         .word  2
label2   .word  3

X        .set   50h

goodsym1 .set   100h + X          : Because value of X is defined before
                                  ; referenced, this is a valid well-defined
                                  ; expression

goodsym2 .set   $                 ; All references to previously defined local
goodsym3 .set   label1            : labels, including the current SPC ($), are
                                  ; considered to be well-defined.

goodsym4 .set   label2 – label1   ; Although label1 and label2 are not
                                  ; absolute symbols, because they are local
                                  ; labels defined in the same section, their
                                  ; difference can be computed by the assembler.
                                  ; The difference is absolute, so the
                                  ; expression is well-defined.
```

(b) Invalid well-defined expressions

```
         .global Y

badsym1  .set   Y                 ; Because Y is external and is not defined in
badsym2  .set   50h + Y           ; the current file, its value is unknown to
                                  ; the assembler. Therefore, it cannot be used
                                  ; where a well-defined expression is needed.

badsym3  .set   50h + Z           ; Although Z is defined in the current file,
Z        .set   60h               ; its definition appears after the expression
                                  ; in which it is used. All symbols and
                                  ; constants which appear in well-defined
                                  ; expressions must be defined before they are
                                  ; referenced.
```

3.9.4 Conditional Expressions

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

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

Conditional expressions evaluate to 1 if true and 0 if false; they 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

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

Relocatable registers can be used in expressions; the addresses of these registers are relocatable with respect to the register section they were defined in, unless they have been declared 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, this 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
```

❑ **Example 1**

The statements in this example use an absolute symbol, LAB1 (which is defined above to have a value of 2). The first statement loads the value 51 into the accumulator:

```
LD  #LAB1  +  ((4+3) *  7), A    ;  ACC A = 51
LD  #LAB1  +  4  +  (3*7), A     ;  ACC A = 27
```

❑ **Example 2**

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

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

```
LD extern_1 – 10, B       ; Legal
LD 10–extern_1, B         ; Can't negate reloc. symbol
LD –(intern_1), B         ; Can't negate reloc. symbol
LD extern_1/10, B         ; / isn't additive operator
LD intern_1 + extern_1, B ; Multiple relocatables
```

❑ **Example 3**

The first statement below is legal; although intern_1 and intern_2 are relocatable, their difference is absolute because they're 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.

```
LD intern_1 – intern_2 + extern_1, B   ; Legal
LD intern_1 + intern_2 + extern_1, B   ; Illegal
```

❑ **Example 4**

An external symbol's placement is important to expression evaluation. Although the statement below is similar to the first statement in the previous example, it is illegal because of left-to-right operator precedence; the assembler attempts to add intern_1 to extern_1.

```
LD intern_1 + extern_1 – intern_2, B   ; 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 –l (lowercase L) option.

Two banner lines, a blank line, and a title line are at the top of each source listing page. Any title supplied by a .title directive is printed on the title line; a page number is printed to the right of the title. If you don't 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 may produce a line in the listing file that shows a source statement number, an SPC value, the object code assembled, and the source statement. A source statement may produce more than one word of object code. The assembler lists the SPC value and object code on a separate line for each additional word. Each additional line is listed immediately following the source statement line.

Field 1: Source Statement Number

Line Number

The source statement number is a decimal. 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

The assembler may precede a line with a letter; the letter indicates that the line is assembled from an included file.

Nesting Level Number

The assembler may precede a line with a number; the number indicates the nesting level of macro expansions or loop blocks.

Field 2: Section Program Counter

This field contains the section program counter (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 by appending one of the following characters to the end of the field:

!	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 3–5 shows an assembler listing with each of the four fields identified.

Example 3–5. Assembler Listing

(a) Mnemonic example

```
      1                         .global RESET, INT0, INT1, INT2
      2                         .global TINT, RINT, XINT, USER
      3                         .global ISR0, ISR1, ISR2
      4                         .global time, rcv, xmt, proc
      5
      6                 initmac .macro
      7                 * initialize macro
      8                         SSBX OVM            ; disable oflow
      9                         LD #0, DP           ; dp = 0
     10                         LD #7, ARP          ; arp = ar7
     11                         LD #037h, A         ; acc = 03fh
     12                         RSBX INTM           ; enable ints
     13                         .endm
     14                 ****************************************
     15                 *       Reset and interrupt vectors      *
     16                 ****************************************
     17  000000                 .sect ”reset”
     18  000000 F073    RESET:  B  init
         000001 0008+
     19  000002 F073    INT0:   B  ISR0
         000003 0000!
     20  000004 F073    INT1:   B  ISR1
         000005 0000!
     21  000006 F073    INT2:   B  ISR2
         000007 0000!
     22
     23                 *
     24  000000                 .sect ”ints”
     25  000000 F073    TINT    B  time
         000001 0000!
     26  000002 F073    RINT    B  rcv
         000003 0000!
     27  000004 F073    XINT    B  xmt
         000005 0000!
     28  000006 F073    USER    B  proc
         000007 0000!
     29                 ****************************************
     30                 *         Initialize processor.        *
     31                 ****************************************
     32  000008         init:   initmac
1                       * initialize macro
1       000008 F7B9             SSBX OVM            ; disable oflow
1       000009 EA00             LD #0, DP           ; dp = 0
1       00000a F4A7             LD #7, ARP          ; arp = ar7
1       00000b E837             LD #037h, A         ; acc = 03fh
1       00000c F6BB             RSBX INTM           ; enable ints
```

Field 1 Field 2 Field 3 Field 4

Example 3–5.Assembler Listing (Continued)

(b) Algebraic example

```
      1                         .global RESET, INT0, INT1, INT2
      2                         .global TINT, RINT, XINT, USER
      3                         .global ISR0, ISR1, ISR2
      4                         .global time, rcv, xmt, proc
      5
      6                 initmac .macro
      7                 * initialize macro
      8                         OVM = 1             ; disable oflow
      9                         DP = #0             ; dp = 0
     10                         ARP = #7            ; arp = ar7
     11                         A = #037h           ; acc = 03fh
     12                         INTM = 0            ; enable ints
     13                         .endm
     14                *****************************************
     15                *       Reset and interrupt vectors       *
     16                *****************************************
     17  000000                 .sect ”reset”
     18  000000 F073    RESET:  goto  init
         000001 0008+
     19  000002 F073    INT0:   goto  ISR0
         000003 0000!
     20  000004 F073    INT1:   goto  ISR1
         000005 0000!
     21  000006 F073    INT2:   goto  ISR2
         000007 0000!
     22
     23                *
     24  000000                 .sect ”ints”
     25  000000 F073    TINT    goto  time
         000001 0000!
     26  000002 F073    RINT    goto  rcv
         000003 0000!
     27  000004 F073    XINT    goto  xmt
         000005 0000!
     28  000006 F073    USER    goto  proc
         000007 0000!
     29                *****************************************
     30                *         Initialize processor.           *
     31                *****************************************
     32  000008         init:   initmac
1                       * initialize macro
1       000008 F7B9             OVM = 1             ; disable oflow
1       000009 EA00             DP = #0             ; dp = 0
1       00000a F4A7             ARP = #7            ; arp = ar7
1       00000b E837             A = #037h           ; acc = 03fh
1       00000c F6BB             INTM = 0            ; enable ints
```

Field 1 Field 2 Field 3 Field 4

3.11 Cross-Reference Listings

A cross-reference listing shows symbols and their definitions. To obtain a cross-reference listing, invoke the assembler with the –x option or use the .option directive. The assembler will append the cross-reference to the end of the source listing.

Example 3–6. Assembler Cross-Reference Listing

```
LABEL                            VALUE     DEFN      REF

INT0                             0002+       14       2
INT1                             0004+       15       2
INT2                             0006+       16       2
ISR0                             REF                      4    14
ISR1                             REF                      4    15
ISR2                             REF                      4    16
RESET                            0000+       13       2
RINT                             0002+       24       3
TINT                             0000+       23       3
VECS                             0006+       26       3
XINT                             0004+       27
init                             0000+       34      13
```

Label column contains each symbol that was defined or referenced during the assembly.

Value column contains a 4-digit hexadecimal number, which is the value assigned to the symbol *or* a name that describes the symbol's attributes. A value may also be 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

Note that when the assembler generates a cross–reference listing for an assembly file that contains .include directives, it keeps a record of the include file and line number in which a symbol is defined/referenced. It does this by assigning a letter reference (A, B, C, etc.) for each include file. The letters are assigned in the order in which the .include directives are encountered in the assembly source file.

For example, the following source files:

(a) incl0.asm

```
.global ABC
nop
nop
```

(b) incl1.asm

```
.global ABC
ld ABC,A
```

(c) incl2.asm

```
.global ABC
stl A,ABC
```

(d) xref.asm

```
.start:
         .include “incl0.asm”
         .include “incl1.asm”
          add #10,A
         .include “incl2.asm”

          nop
          nop
          b start

          .global start

           .bss ABC,2
```

produce the following cross–reference listing:

```
xref.asm                        PAGE    1

 1 000000       start:
 2                   .include ”incl0.asm”
 3                   .include ”incl1.asm”
 4 000003 F000       add #10,A
   000004 000A
 5                   .include ”incl2.asm”
 6
 7 000006 F495       nop
 8 000007 F495       nop
 9 000008 F073       b start
   000009 0000’
10
11                   .global start
12
13 000000            .bss ABC,2

xref.asm                        PAGE    2

LABEL            VALUE       DEFN    REF

.TMS320C540      000001       0
.TMS320C541      000000       0
.TMS320C542      000000       0
.TMS320C543      000000       0
.TMS320C544      000000       0
.TMS320C545      000000       0
.TMS320C545LP    000000       0
.TMS320C546      000000       0
.TMS320C546LP    000000       0
.TMS320C548      000000       0
ABC              000000–     13    A  1  B  1  B  2
                                    C  1  C  2
__far_mode       000000       0
__lflags         000000       0
start            000000’      1       9    11
```

The A in the cross–reference listing refers to incl0.asm (the first file included). B refers to incl1.asm; C refers to incl2.asm.

Chapter 4

Assembler Directives

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

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

This chapter is divided into two parts: the first part (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 Directives Summary

This section summarizes the assembler directives. The summaries are grouped into the following categories:

- Directives that define sections
- Directives that initialize constants (data and memory
- Directives that align the section program counter (SPC)
- Directives that format the output listing
- Directives that reference other files
- Directives that control conditional assembly
- Directives that define symbols at assembly time
- Directives that perform miscellaneous functions

Besides the assembler directives documented here, the TMS320C54x C compiler uses several directives for symbolic debugging and absolute listing. 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. Absolute listing directives are not entered by the user but are inserted into the source program by the absolute lister. They are discussed in Chapter 10, *Absolute Lister Description*.

Note: Labels and Comments in Syntax

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 thing that should appear in the first column) and comments should be preceded by a semicolon or a star if the comment is the only statement on the line. To improve readability, labels and comments are not shown as part of the directive syntax.

Table 4–1. Directives That Define Sections

Mnemonic and Syntax	Description	Page
.bss *symbol*, *size in words* [*, blocking*] [**,** *alignment*]	Reserve *size* words in the .bss (uninitialized data) section	4-30
.data	Assemble into the .data (initialized data) section	4-37
.sect "*section name***"**	Assemble into a named (initialized) section	4-70
.text	Assemble into the .text (executable code) section	4-81
symbol **.usect "***section name***",** *size in words* [**,** *blocking*] [,*alignment flag*]	Reserve *size* words in a named (uninitialized) section	4-83

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

Mnemonic and Syntax	Description	Page
.bes *size in bits*	Reserve *size* bits in the current section; note that a label points to the last addressable word in the reserved space	4-73
.byte *value*$_1$ [**,** *...* **,** *value*$_n$]	Initialize one or more successive bytes in the current section	4-33
.field *value* [**,** *size in bits*]	Initialize a variable-length field	4-43
.float *value* [**,** *...* **,** *value*$_n$]	Initialize one or more 32-bit, IEEE single-precision, floating-point constants	4-46
.int *value*$_1$ [**,** ... **,** *value*$_n$]	Initialize one or more 16-bit integers	4-52
.long *value*$_1$ [**,** ... **,** *value*$_n$]	Initialize one or more 32-bit integers	4-57
.space *size in bits;*	Reserve *size* bits in the current section; note that a label points to the beginning of the reserved space	4-73
.string "*string*$_1$**"** [**,** ... **,** "*string*$_n$**"**]	Initialize one or more text strings	4-76
.pstring "string$_1$**"** [, ... ,"*string*$_n$"]	Initialize one or more text strings (packed).	4-76
.xfloat *value*$_1$ [, ...,*value*$_n$]	Initialize one or more 32-bit integers, IEEE single-precision, floating-point constants, but do not align on long word boundary.	4-46
.xlong *value*$_1$ [, ...,*value*$_n$]	Initialize one or more 32-bit integers, but do not align on long word boundary.	4-57
.word *value*$_1$ [, ... , *value*$_n$]	Initialize one or more 16-bit integers.	4-52

Table 4–3. Directives That Align the Section Program Counter (SPC)

Mnemonic and Syntax	Description	Page
.align [*size in words*]	Align the SPC on a word boundary specified by the parameter, which must be a power of 2, or default to page boundary. Aligns to word (1), even (2), etc.	4-27

Table 4–4. Directives That Format the Output Listing

Mnemonic and Syntax	Description	Page
.drlist	Enable listing of all directive lines (default)	4-38
.drnolist	Suppress listing of certain directive lines	4-38

Mnemonic and Syntax	Description	Page
.fclist	Allow false conditional code block listing (default)	4-42
.fcnolist	Suppress false conditional code block listing	4-42
.length *page length*	Set the page length of the source listing	4-54
.list	Restart the source listing	4-55
.mlist	Allow macro listings and loop blocks (default)	4-61
.mnolist	Suppress macro listings and loop blocks	4-61
.nolist	Stop the source listing	4-55
.option {**B** | **L** | **M** | **R** | **T** | **W** | **X**}	Select output listing options	4-66
.page	Eject a page in the source listing	4-68
.sslist	Allow expanded substitution symbol listing	4-74
.ssnolist	Suppress expanded substitution symbol listing (default)	4-74
.tab *size*	Set tab size	4-80
.title "*string***"**	Print a title in the listing page heading	4-82
.width *page width*	Set the page width of the source listing	4-54

Table 4–5. Directives That Reference Other Files

Mnemonic and Syntax	Description	Page
.copy ["]*filename*["]	Include source statements from another file	4-34
.def $symbol_1$ [**,** ... **,** $symbol_n$]	Identify one or more symbols that are defined in the current module and may be used in other modules	4-47
.global $symbol_1$ [**,** ... **,** $symbol_n$]	Identify one or more global (external) symbols	4-47
.include ["]*filename*["]	Include source statements from another file	4-34
.mlib ["]*filename*["]	Define macro library	4-59
.ref $symbol_1$ [**,** ... **,** $symbol_n$]	Identify one or more symbols that are used in the current module but may be defined in another module	4-47

Table 4–6. Directives That Control Conditional Assembly

Mnemonic and Syntax	Description	Page
.break [*well-defined expression*]	End .loop assembly if condition is true. The .break construct is optional.	4-58
.else *well-defined expression*	Assemble code block if the .if condition is false. The .else construct is optional.	4-50
.elseif *well-defined expression*	Assemble code block if the .if condition is false and the .elseif condition is true. The .elseif construct is optional.	4-50
.endif	End .if code block	4-50
.endloop	End .loop code block	4-58
.if *well-defined expression*	Assemble code block if the condition is true	4-50
.loop [*well-defined expression*]	Begin repeatable assembly of a code block	4-58

Table 4–7. Directives That Define Symbols at Assembly Time

Mnemonic and Syntax	Description	Page
.asg [**"**]*character string*[**"**], *substitution symbol*	Assign a character string to a substitution symbol	4-28
.endstruct	End structure definition	4-77
.equ	Equate a value with a symbol	4-72
.eval *well-defined expression*, *substitution symbol*	Perform arithmetic on numeric substitution symbols	4-28
.label *symbol*	Define a load-time relocatable label in a section	4-53
.set	Equate a value with a symbol	4-72
.struct	Begin structure definition	4-77
.tag	Assign structure attributes to a label	4-77

Table 4–8. Miscellaneous Directives

Mnemonic and Syntax	Description	Page
.algebraic	Signifies that the file contains algebraic assembly source	4-26
.emsg *string*	Send user-defined error messages to the output device	4-39
.end	End program	4-41
.mmregs	Enter memory-mapped registers into the symbol table	4-62
.mmsg *string*	Send user-defined messages to the output device	4-39
.newblock	Undefine local labels	4-65
.sblock [**"**]*section name*[**"**] [, ... , "*section name*"]	Designates sections for blocking	4-69
.version [*value*]	Specify the device for which processor instructions are being built	4-86
.wmsg *string*	Send user-defined warning messages to the output device	4-39

4.2 Compatibility With the TMS320C1x/C2x/C2xx/C5x Assembler Directives

This section explains how the TMS320C54x assembler directives differ from the TMS320C1x/C2x/C2xx/C5x assembler directives.

- The ’C54x .long and .float directives place the *most* significant word of the value at the lower address, while the ’C1x/C2x/C2xx/C5x assembler directives place the *least* significant word at the lower address. Also, the ’C54x .long and .float directives automatically align the SPC on an even word boundary, while the C1x/C2x/C2xx/C5x assembler directives do not.

- Without arguments, the ’C54x and the ’C1x/C2x/C2xx/C5x assemblers both align the SPC at the next 128-word boundary. However, the ’C54x .align directive also accepts a constant argument, which must be a power of 2, and this argument causes alignment of the SPC on that word boundary. The .align directive for the ’C1x/C2x/C2xx/C5x assembler does not accept this constant argument.

- The .even directive, which causes alignment at word boundaries when used with the ’C1x/C2x/C2xx/C5x assembler, is not supported by the ’C54x assembler. The .align directive with an argument of 1 replaces the .even directive.

- The .field directive for the ’C54x packs fields into words starting at the *most* significant bit of the word. The ’C1x/C2x/C2xx/C5x assembler .field directive places fields into words starting at the *least* significant bit of the word.

- The .field directive for the ’C54x handles values of 1 to 32 bits, contrasted with the ’C1x/C2x/C2xx/C5x assembler which handles values of 1 to 16 bits. With the ’C54x assembler, objects that are 16 bits or larger start on a word boundary and are placed with the most significant bits at the lower address.

- The ’C54x .bss and .usect directives have an additional flag called the alignment flag which specifies alignment on an even word boundary. The ’C1x/C2x/C2xx/C5x .bss and .usect directives do not use this flag.

- The .string directive for the ’C54x initializes one character per word, unlike the ’C1x/C2x/C2xx/C5x assembler .string, which packs two characters per word. The new .pstring directive packs two characters per word, as the former .string did.

- The following directives are new with the ’C54x assembler and are not supported by the ’C1x/C2x/C2xx/C5x assembler:

Directive	Usage
.xfloat	Same as .float without automatic alignment
.xlong	Same as .long without automatic alignment
.pstring	Same as .string, but packs two chars/word

4.3 Directives That Define Sections

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

- **.bss** reserves space in the .bss section for uninitialized variables.
- **.data** identifies portions of code in the .data section. The .data section usually contains initialized data.
- **.sect** defines initialized named sections and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- **.text** identifies portions of code in the .text section. The .text section usually contains executable code.
- **.usect** 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 0000                 .text
 5 0000 0001            .word   1,2
   0001 0002
 6 0002 0003            .word   3,4
   0003 0004
 7
 8              ************************************************
 9              *     Start assembling into the .data section    *
10              ************************************************
11 0000                 .data
12 0000 0009            .word   9, 10
   0001 000A
13 0002 000B            .word   11, 12
   0003 000C
14
15              ************************************************
16              *       Start assembling into a named,           *
17              *       initialized section, var_defs            *
18              ************************************************
19 0000                 .sect   ”var_defs”
20 0000 0011            .word   17, 18
   0001 0012
21
22              ************************************************
23              *    Resume assembling into the .data section    *
24              ************************************************
25 0004                 .data
26 0004 000D            .word   13, 14
   0005 000E
27 0000                 .bss    sym, 19   ; Reserve space in .bss
28 0006 000F            .word   15, 16    ; Still in .data
   0007 0010
29
30              ************************************************
31              *    Resume assembling into the .text section    *
32              ************************************************
33 0004                 .text
34 0004 0005            .word   5, 6
   0005 0006
35 0000        usym     .usect  ”xy”, 20   ; Reserve space in xy
36 0006 0007            .word   7, 8       ; Still in .text
   0007 0008
```

4.4 Directives That Initialize Constants

Several directives assemble values for the current section:

- The **.bes** and **.space** directives reserve a specified number of bits in the current section. The assembler fills these reserved bits with 0s.

  You can reserve a specified number of words by multiplying the number of bits by 16.

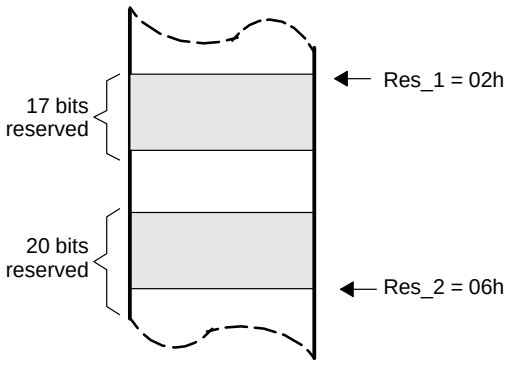
  - When you use a label with .space, it points to the *first* word that contains reserved bits.
  - When you use a label with .bes, it points to the *last* word that contains reserved bits.

  Figure 4–1 shows the .space and .bes directives. Assume the following code has been assembled for this example:

```
1
2              ** .space and .bes directives
3
4 0000 0100             .word           100h, 200h
  0001 0200
5 0002         Res_1:   .space          17
6 0004 000f             .word           15
7 0006         Res_2:   .bes            20
8 0007 00ba             .byte           0BAh
```

  Res_1 points to the first word in the space reserved by .space. Res_2 points to the last word in the space reserved by .bes.

Figure 4–1. The .space and .bes Directives



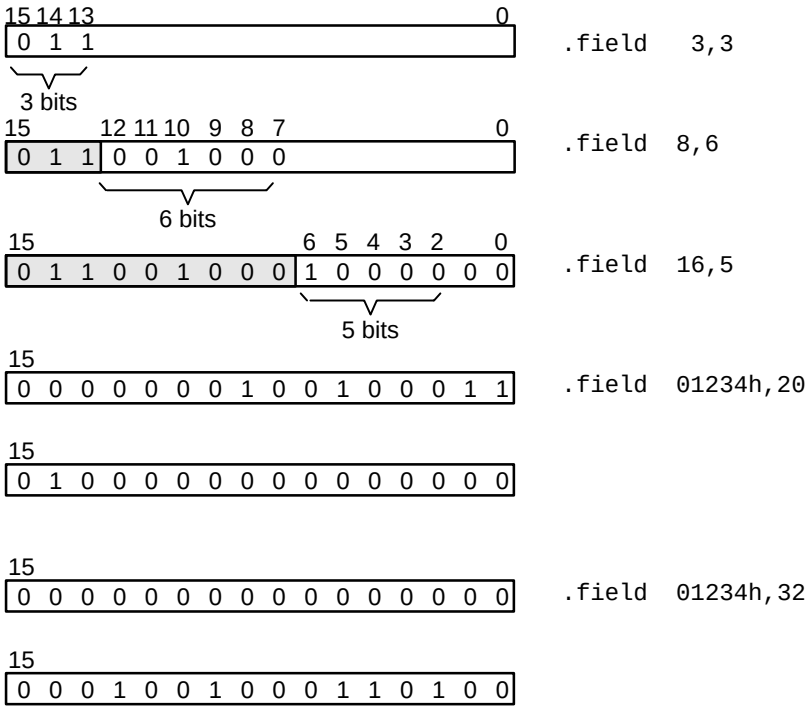
- **.byte** places one or more 8-bit values into consecutive words of the current section. This directive is similar to .word, except that the width of each value is restricted to 8 bits.

- 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. For this example, assume the following code has been assembled; notice that the SPC doesn't change for the first three fields (the fields are packed into the same word):

```
4 0000 6000          .field          3, 3
5 0000 6400          .field          8, 6
6 0000 6440          .field          16, 5
7 0001 0123          .field          01234h,20
  0002 4000
8 0003 0000          .field          01234h,32
  0004 1234
```

Figure 4–2. The .field Directive



- **.float** and **.xfloat** calculate the single-precision (32-bit) IEEE floating-point representation of a single floating-point value and store it in two consecutive words in the current section. The most significant word is stored first. The .float directive automatically aligns to the long word boundary, and .xfloat does not.

- **.int** and **.word** place one or more 16-bit values into consecutive words in the current section.

- **.long** and **.xlong** place 32-bit values into consecutive 2-word blocks in the current section. The most significant word is stored first. The .long directive automatically aligns to a long word boundary, and the .xlong directive does not.

- **.string** and **.pstring** place 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. The .pstring also has a width of 8 bits but packs two characters into a word. For .pstring, the last word in a string is padded with null characters (0) if necessary.

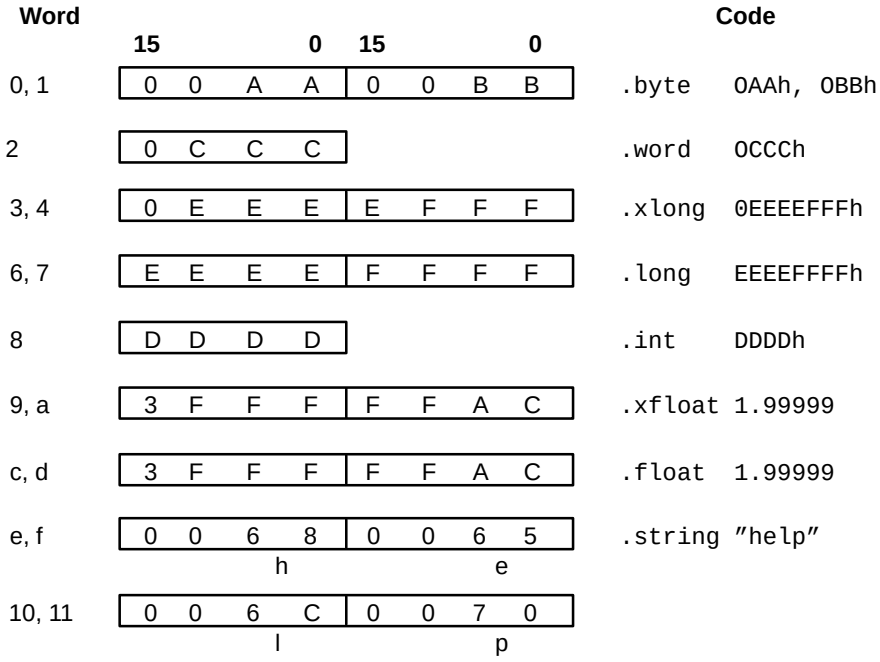
Note: The .byte, .word, .int, .long, .xlong, .string, .pstring, .float, .xfloat, and .field Directives in a .struct/.endstruct Sequence

The .byte, .word, .int, .long, .xlong, .string, .pstring, .float, .xfloat. 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 Section 4.9, *Assembly-Time Symbol Directives*, on page 4-21.

Figure 4–3 compares the .byte, .int, .long, .xlong, .float, .xfloat, .word, and .string directives. For this example, assume that the following code has been assembled:

```
1 0000 00aa          .byte           0AAh, 0BBh
  0001 00bb
2 0002 0ccc          .word           0CCCh
3 0003 0eee          .xlong          0EEEEFFFh
  0004 efff
4 0006 eeee          .long           0EEEEFFFFh
  0007 ffff
5 0008 dddd          .int            0DDDDh
6 0009 3fff          .xfloat         1.99999
  000a ffac
7 000c 3fff          .float          1.99999
  000d ffac
8 000e 0068          .string         "help"
  000f 0065
  0010 006c
  0011 0070
```

Figure 4–3. Initialization Directives



4.5 Directives That Align the Section Program Counter

The **.align** directive aligns the SPC at a 1-word to 128-word boundary. This ensures that the code following the directive begins on an x-word or page boundary. 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^{16} (although directives beyond 2^7 are not meaningful). For example:

Operand of	1	aligns SPC to word boundary
	2	aligns SPC to long word/even boundary
	128	aligns SPC to page boundary

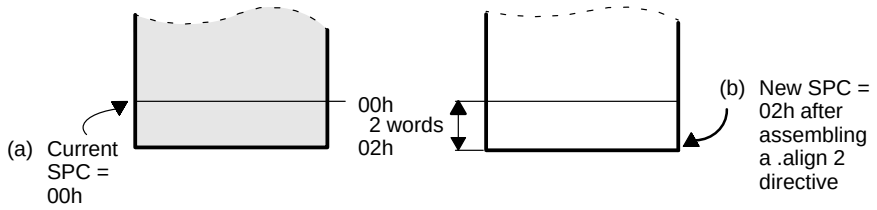
The .align directive with no operands defaults to 128, that is, to a page boundary.

Figure 4–4 demonstrates the .align directive. Assume that the following code has been assembled:

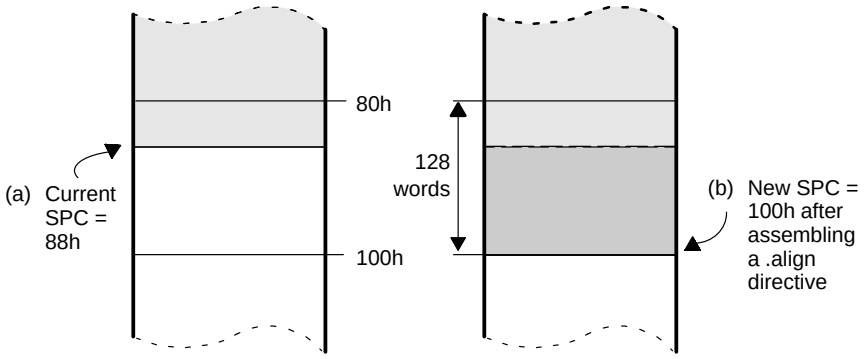
```
1 0000 4000          .field          2, 3
2 0000 4160          .field          11, 8
3                    .align          2
4 0002 0045          .string         "Errorcnt"
  0003 0072
  0004 0072
  0005 006f
  0006 0072
  0007 0063
  0008 006e
  0009 0074
5                    .align
6 0100 0004          .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. 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 contains a listing of 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 stop the assembler from printing selected source statements in the listing file. Use the .list directive to turn the listing on again.

- The source code contains a listing of 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:

  B limits the listing of .byte directives to one line.

  L limits the listing of .long directives to one line.

  M turns off macro expansions in the listing.

  R resets the B, M, T, and W options.

  T limits the listing of .string directives to one line.

  W limits the listing of .word directives to one line.

  X produces a symbol cross-reference listing. (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 **.sslist** and **.ssnolist** directives allow and suppress substitution symbol expansion 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:

- 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 subsection 2.8.1, *External Symbols*, on page 2-18.) 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 only if it 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 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 Conditional Assembly Directives

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

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

.if *expression*	marks the beginning of a conditional block and assembles code if the .if condition is true.
.elseif *expression*	marks a block of code to be assembled if the .if condition is false and .elseif is true.
.else	marks a block of code to be assembled if the .if condition is false.
.endif	marks the end of a conditional block and terminates the block.

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

.loop *expression*	marks the beginning a repeatable block of code.
.break *expression*	tells the assembler to continue to repeatedly assemble when the .break expression is false, and to go to the code immediately after .endloop when the expression is true.
.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 subsection 3.9.4, *Conditional Expressions*, on page 3-26.

4.9 Assembly-Time Symbol Directives

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

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

```
.asg   ”10, 20, 30, 40”, coefficients

.byte  coefficients
```

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

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

- The **.label** directive defines a special symbol that refers to the 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.

- The **.set** and **.equ** directives set a constant value to a symbol. The symbol is stored in the symbol table and cannot be refined. For example:

```
bval   .set   0100h
       .byte  bval, bval*2, bval+12
       B      bval
```

  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 then 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's used.

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

COORD .tag type         ; declare COORD (coordinate)

ADD   COORD.Y, A

      .bss COORD, T_LEN ; actual memory allocation
```

4.10 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.algebraic** directive tells the assembler that the file contains algebraic assembly source code. This must be the first line in the file if the –mg assembler option is not used.
- The **.end** directive terminates assembly. It should be the last source statement of a program. This directive has the same effect as an end-of-file.
- The **.mmregs** directive defines symbolic names for the memory-mapped register. Using .mmregs is the same as executing a .set for all memory-mapped registers. See Table 4–9 on page 4-62 for a list of memory-mapped registers.
- The **.newblock** directive resets local labels. Local labels are symbols of the form $n or name?. They are defined when they appear in the label field. Local labels are temporary labels that can be used as operands for jump instructions. The .newblock directive limits the scope of local labels by resetting them after they are used. For more information about local labels, see subsection 3.8.6, *Local Labels*, on page 3-19.
- The **.sblock** directive designates sections for blocking. Blocking is an address alignment mechanism similar to page alignment, but weaker. A blocked section is guaranteed not to cross a page boundary (128 words) if it is smaller than a page, or to start on a page boundary if it is larger than a page. Note that this directive allows specification of blocking for initialized sections only, not uninitialized sections declared with .usect or the .bss section. The section names may or may not be enclosed in quotes.
- The **.version** directive determines the processor for which instructions are being built. Each ’C54x device has its own value.

These 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.

4.11 Directives Reference

The remainder of this chapter is a reference. Generally, the directives are organized alphabetically, one directive per page; however, related directives (such as .if/.else/.endif) are presented together on one page. Following is an alphabetical table of contents for the directives reference:

Syntax

```
.algebraic
```

Description

The **.algebraic** directive tells the assembler that this file contains algebraic assembly source code. This directive must be the first line in the file if the –mg option is not used.

> **Note: Mixing Algebraic and Mnemonic Assembly Code**
>
> Algebraic and mnemonic assembly code cannot be mixed within the same source file.

The **.algebraic** directive does not provide a method for mixing algebraic and mnemonic statements within the same source file. It provides a means of selecting algebraic assembly without specifying the –mg assembler option.

Syntax

.align [*size in words*]

Description

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size in words* parameter. The *size* may be any power of 2, although only certain values are useful for alignment. An operand of 128 aligns the SPC on the next page boundary, and this is the default if no *size* is given. The assembler assembles words containing null values (0) up to the next x-word boundary.

Operand of	1	aligns SPC to word boundary
	2	aligns SPC to long word/even boundary
	128	aligns SPC to page boundary

Using the .align directive has two effects:

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

Example

This example shows several types of alignment, including .align 2, .align 4, and a default .align.

```
 1 0000 0004       .byte     4
 2                 .align    2
 3 0002 0045       .string   "Errorcnt"
   0003 0072
   0004 0072
   0005 006F
   0006 0072
   0007 0063
   0008 006E
   0009 0074
 4                 .align
 5 0080 6000       .field    3,3
 6 0080 6A00       .field    5,4
 7                 .align    2
 8 0082 6000       .field    3,3
 9                 .align    8
10 0088 5000       .field    5,4
11                 .align
12 0100 0004       .byte     4
```

Syntax

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

Description

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

- The assembler assigns the *character string* to the substitution symbol. The quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the substitution symbol.
- The *substitution symbol* is a required parameter that must be a valid symbol name. The substitution symbol may be 32 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

The **.eval** directive performs arithmetic on substitution symbols, which are stored in the substitution symbol table. This directive evaluates the 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 consisting of legal values that have been previously defined, so that the result is an absolute.

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

```
       1                        .sslist;show expanded sub. symbols
       2              *
       3              *         .asg/.eval example
       4              *
       5                        .asg    *+, INC
       6                        .asg    AR0, FP
       7
       8 0000 f000              ADD     #100, A
         0001 0064
       9 0002 6d90              MAR     *FP+
#                               MAR     *AR0+
      10 0003 6d90              MAR     *AR0+
      11
      12                        .asg    0, x
      13                        .loop   5
      14                        .eval   x+1, x
      15                        .word   x
      16                        .endloop
1                               .eval   x+1, x
#                               .eval   0+1, x
1       0004 0001               .word   x
#                               .word   1
1                               .eval   x+1, x
#                               .eval   1+1, x
1       0005 0002               .word   x
#                               .word   2
1                               .eval   x+1, x
#                               .eval   2+1, x
1       0006 0003               .word   x
#                               .word   3
1                               .eval   x+1, x
#                               .eval   3+1, x
1       0007 0004               .word   x
#                               .word   4
1                               .eval   x+1, x
#                               .eval   4+1, x
1       0008 0005               .word   x
#                               .word   5
```

Syntax

.bss *symbol size in words* [**,** *blocking flag*] [**,** *alignment flag*]

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 should correspond to the variable that you're reserving space for.
- The *size in words* is a required parameter; it must be an absolute expression. The assembler allocates *size* words in the .bss section. There is no default size.
- The *blocking flag* is an optional parameter. If you specify a value greater than 0 for this parameter, the assembler allocates *size* words contiguously. This means that the allocated space will not cross a page boundary unless size is greater than a page, in which case, the object will start on a page boundary.
- The *alignment flag* is an optional parameter. This flag causes the assembler to allocate size on long word boundaries.

The assembler follows two rules when it allocates space in the .bss section:

Rule 1 Whenever a hole is left in memory (as shown in Figure 4–5), the .bss directive attempts to fill it. When a .bss directive is assembled, the assembler searches its list of holes left by previous .bss directives and tries to allocate the current block into one of the holes. (This is the standard procedure whether the contiguous allocation option has been specified or not.)

Rule 2 If the assembler does not find a hole large enough to contain the requested space, it checks to see whether the blocking option is requested.

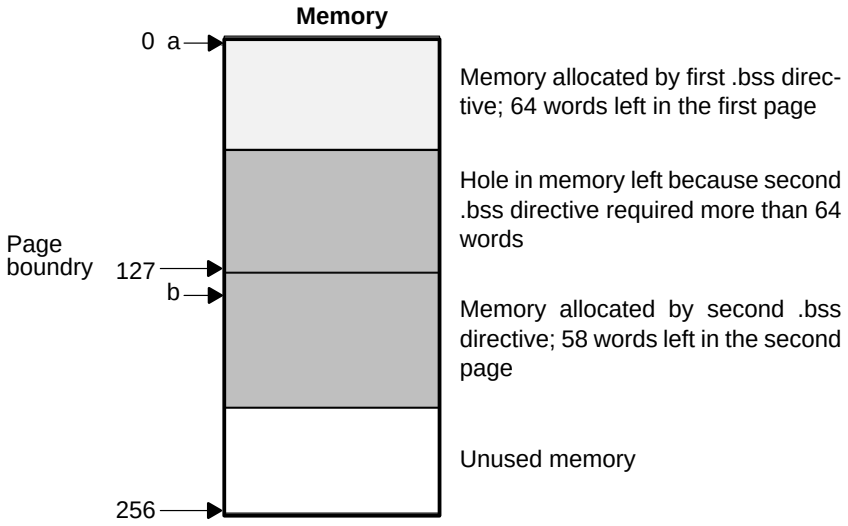
- If you do not request blocking, the memory is allocated at the current SPC.
- If you request blocking, the assembler checks to see whether there is enough space between the current SPC and the page boundary. If there is not enough space, the assembler creates another hole and allocates the space on the next page.

The blocking option allows you to reserve up to 128 words in the .bss section and ensure that they fit on one page of memory. (Of course, you can reserve more than 128 words at a time, but they cannot fit on a single page.) The following example code reserves two blocks of space in the .bss section.

```
memptr:    .bss     A,64,1
memptr1:   .bss     B,70,1
```

Each block must be contained within the boundaries of a single page; after the first block is allocated, however, the second block cannot fit on the current page. As Figure 4–5 shows, the second block is allocated on the next page.

Figure 4–5. Allocating .bss Blocks Within a Page



Section directives for initialized sections (.text, .data, and .sect) end the current section and begin assembling into another section. The .bss directive, however, does not affect the current section. The assembler assembles the .bss directive and then resumes assembling code into the current section. For more information about COFF sections, see Chapter 2, *Introduction to Common Object File Format*.

Example

In this example, the .bss directive is used to allocate space for two variables, TEMP and ARRAY. The symbol TEMP points to 4 words of uninitialized space (at .bss SPC = 0). The symbol ARRAY points to 100 words of uninitialized space (at .bss SPC = 04h); this space must be allocated contiguously within a page. Note that symbols declared with the .bss directive can be referenced in the same manner as other symbols and can also be declared external.

```
 1                ****************************************
 2                **  Assemble into the .text section.  **
 3                ****************************************
 4 0000                   .text
 5 0000 e800              LD      #0, A
 6                ****************************************
 7                **  Allocate 4 words in .bss for TEMP.  **
 8                ****************************************
 9 0000           Var_1:  .bss    TEMP, 4
10
11                ****************************************
12                **           Still in .text           **
13                ****************************************
14 0001 f000             ADD     #56h, A
   0002 0056
15 0003 f066             MPY     #73h, A
   0004 0073
16
17                ****************************************
18                **  Allocate 100 words in .bss for the  **
19                **  symbol named ARRAY; this part of    **
20                **  .bss must fit on a single page.     **
21                ****************************************
22 0004                  .bss    ARRAY, 100, 1
23
24                ****************************************
25                **  Assemble more code into .text.      **
26                ****************************************
27 0005 8000-            STL     A, Var_1
28
29                ****************************************
30                **  Declare external .bss symbols.      **
31                ****************************************
32                       .global ARRAY, TEMP
33                       .end
```

Syntax

.byte *value*$_1$ [**,** ... **,** *value*$_n$]

Description

The **.byte** directive places one or more bytes into consecutive words of the current section. Each byte is placed in a word by itself; the 8 MSBs are filled with 0s. A *value* can be either:

- An expression that the assembler evaluates and treats as an 8-bit signed number, or
- A character string enclosed in double quotes. Each character in a string represents a separate value.

Values are not packed or sign-extended; each byte occupies the 8 least significant bits of a full 16-bit word. The assembler truncates values greater than 8 bits. You can use up to 100 value parameters, but the total line length cannot exceed 200 characters.

If you use a label, it points to the location where the assembler places the first byte.

Note that when you use .byte in a .struct/.endstruct sequence, .byte defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, see Section 4.9, *Assembly-Time Symbol Directives*, on page 4-21.

Example

In this example 8-bit values (10, –1, abc, and a) are placed into consecutive words in memory. The label strx has the value 100h, which is the location of the first initialized word.

```
1 0000                .space   100h * 16
2 0100 000a  STRX     .byte    10, -1, "abc", 'a'
  0101 00ff
  0102 0061
  0103 0062
  0104 0063
  0105 0061
```

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. 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 may be enclosed in double quotes and must follow operating system conventions. You can specify a full pathname (for example, c:\dsp\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 environment variable A_DIR.

For more information about the –i option and A_DIR, see Section 3.4, *Naming Alternate Directories for Assembler Input*, on page 3-6.

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 (note listing below). The copy file byte.asm contains a .copy statement for a second file, word.asm.

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

copy.asm (source file)	**byte.asm (first copy file)**	**word.asm (second copy file)**

`.space 29`
`.copy "byte.asm"`
 	
`**Back in original file`
`.pstring "done"`
	`** In byte.asm`
	`.byte 32,1+ 'A'`
	`.copy "word.asm"`
	`** Back in byte.asm`
	`.byte 67h + 3q`
		`** In word.asm`
		`.word 0ABCDh, 56q`

Listing file:

```
      1 0000           .space 29
      2                .copy "byte.asm"
A     1              ** In byte.asm
A     2 0002 0020      .byte 32,1+ 'A'
        0003 0042
A     3                .copy "word.asm"
B     1              * In word.asm
B     2 0004 ABCD      .word 0ABCDh, 56q
        0005 002E
A     4              ** Back in byte.asm
A     5 0006 006A      .byte 67h + 3q
      3
      4              ** Back in original file
      5 0007 646F      .pstring "done"
        0008 6E65
```

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)

```
   .space  29
   .include "byte2.asm"

**Back in original file
   .string "done"
```

byte2.asm (first include file)

```
   ** In byte2.asm
      .byte 32,1+ 'A'
      .include "word2.asm"
   ** Back in byte2.asm
      .byte  67h + 3q
```

word2.asm (second include file)

```
   ** In word2.asm
      .word 0ABCDh, 56q
```

Listing file:

```
1 0000           .space 29
2                .include "byte2.asm"
3
4             ** Back in original file
5 0007 0064      .string "done"
  0008 006F
  0009 006E
  000a 0065
```

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. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you use a section control directive.

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 0000                  .data
 5 0000                  .space          0CCh
 6
 7                ****************************************
 8                **  Assemble into .text.              **
 9                ****************************************
10 0000                  .text
                        ; constant into .data.
11      0000  INDEX     .set            0
12 0000 e800            LD              #INDEX, A
13
14               ****************************************
15               **  Assemble into .data.              **
16               ****************************************
17 000d          Table:  .data
18 000d ffff            .word   –1   ; Assemble 16–bit
19
20 000e 00ff            .byte   0FFh ; Assemble 8–bit
21                                   ; constant into .data.
22
23               ****************************************
24               **  Assemble into .text.              **
25               ****************************************
26 0001                 .text
27 0001 000d”           ADD     Table, A
28
29               ****************************************
30               **  Resume assembling into the .data  **
31               **  section at address 0Fh.           **
32               ****************************************
33 000f                 .data
```

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:

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

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

Example

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

Source file:

```
     .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
```

Syntax

.emsg *string*
.mmsg *string*
.wmsg *string*

Description

These directives allow you to define your own error and warning messages. 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 error messages 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 assembly-time messages to the standard output device in the same manner as the .emsg and .wmsg directives, but 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 warning messages to the standard output device in the same manner as the .emsg directive, but 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:

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

          MSG_EX PARAM

          MSG_EX
```

Listing file:

```
       1                        .global PARAM
       2              MSG_EX    .macro  parm1
       3                        .if     $symlen(parm1) = 0
       4                        .emsg   "ERROR -- MISSING
PARAMETER"
       5                        .else
       6                        add     parm1, A
       7                        .endif
       8                        .endm
       9
      10 0000                   MSG_EX PARAM
1                               .if     $symlen(parm1) = 0
1                               .emsg   "ERROR -- MISSING
PARAMETER"
1                               .else
1        0000 0000!             add     PARAM, A
1                               .endif
      11
      12 0001                   MSG_EX
1                               .if     $symlen(parm1) = 0
1                               .emsg   "ERROR -- MISSING
PARAMETER"
  ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
1                               .else
1                               add     parm1, A
1                               .endif

 1 Error, No Warnings
```

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

```
TMS320C54x COFF Assembler       Version x.xx
Copyright (c) 1993-1994    Texas Instruments Incorporated
 PASS 1
 PASS 2
***  ERROR!    line12:   ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
                  .emsg "ERROR -- MISSING PARAMETER"


 1 Error,  No Warnings

Errors in source - Assembler Aborted
```

Syntax

```
.end
```

Description

The **.end** directive is optional and terminates assembly. It should be the last source statement of a program. The assembler ignores any source statements that follow a .end directive.

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

Note: Ending a Macro

Use .endm to end a macro.

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
        ABS     A
        ADD     #TEMP, A
        STL     A, LOC1
        .end
        .byte   4
        .word   CCCh
```

Listing file:

```
1 0000       START:  .space  300
2      000F  TEMP    .set    15
3 0000               .bss    LOC1, 48h
4 0013 F485          ABS     A
5 0014 F000          ADD     #TEMP, A
  0015 000F
6 0016 8000–         STL     A, LOC1
7                    .end
```

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   #1024, A
        .else
        ADD   #1024*10, A
        .endif
        .fcnolist
        .if   AAA
        ADD   #1024, A
        .else
        ADD   #1024*10, A
        .endif
```

Listing file:

```
        1      0001  AAA    .set  1
        2      0000  BBB    .set  0
        3                   .fclist
        4
        5                   .if   AAA
        6 0000 F000         ADD   #1024, A
          0001 0400
        7                   .else
        8                   ADD   #1024*10, A
        9                   .endif
       10
       11                   .fcnolist
       12
       14 0002 F000         ADD   #1024, A
          0003 0400
```

Syntax

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

Description

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

- The *value* is a required parameter; it is an expression that is evaluated and placed in the field. If the value is relocatable, *size* must be 16.
- The *size* 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 16 bits. If you specify a size of 16 or more, the field will start on a word boundary. If you specify a value that cannot fit into *size* bits, 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 – value truncated.
```

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, increments the SPC, and begins packing fields into the next word. You can use the .align directive with an operand of 1 to force the next .field directive to begin packing into a new word.

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

When you use .field in a .struct/.endstruct sequence, .field defines a member's size; it does not initialize memory. For more information about .struct/ .endstruct, see Section 4.9, *Assembly-Time Symbol Directives*, on page 4-21.

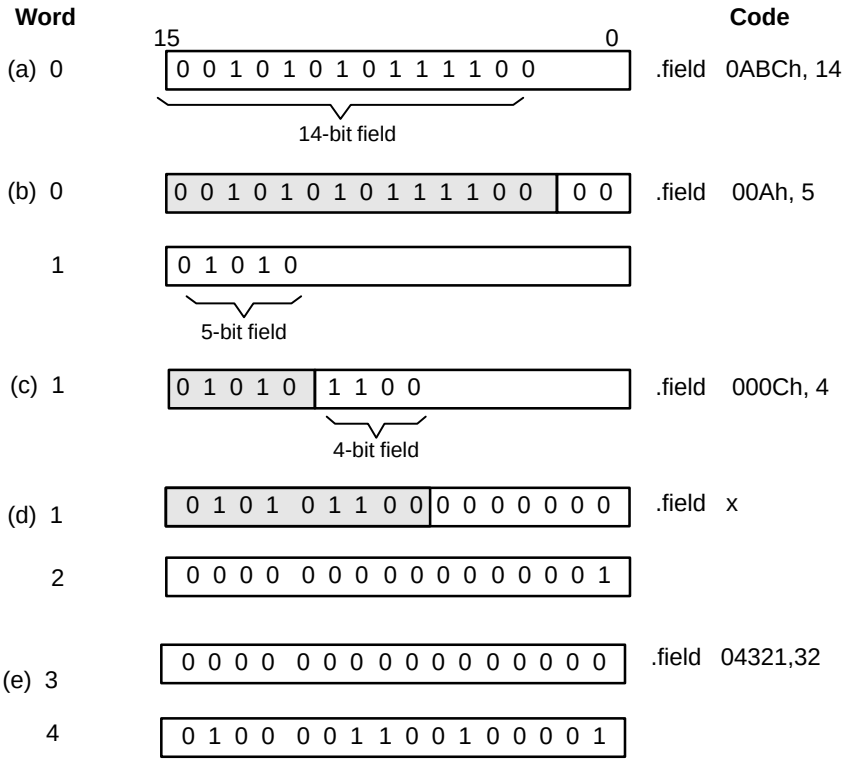
Example

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 0000 2AF0                 .field  0ABCh, 14
 5
 6                    **********************************
 7                    **    Initialize a 5-bit field    **
 8                    **         in a new word.         **
 9                    **********************************
10 0001 5000  L_F:     .field  0Ah, 5
11
12                    *********************************
13                    **    Initialize a 4-bit field    **
14                    **        in the same word.       **
15                    **********************************
16 0001 5600  x:       .field  0Ch, 4
17
18                    **********************************
19                    **    16-bit relocatable field    **
20                    **        in the next word.       **
21                    **********************************
22 0002 0001'               .field  x
23
24                    **********************************
25                    **   Initialize a 32-bit field.   **
26                    **********************************
27 0003 0000                .field  04321h, 32
   0004 4321
```

Figure 4–6 shows how the directives in this example affect memory.

Figure 4–6. The .field Directive



Syntax

.float *value*$_1$ [**,** ... **,** *value*$_n$]
.xfloat *value*$_1$ [**,** ... **,** *value*$_n$]

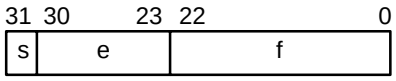
Description

The **.float** and **.xfloat** directives place the floating-point representation of one or more floating-point constants into the current section. The *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 the long-word boundaries unless the .xfloat directive is used. The .xfloat directive performs the same function as the .float directive but does not align the result on the long word boundary.

The 32-bit value consists of three fields:

Field	Meaning
s	A 1-bit sign field
e	An 8-bit biased exponent
f	A 23-bit fraction

The value is stored most significant word first, least significant word second, in the following format:



When you use .float in a .struct/.endstruct sequence, .float defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, see Section 4.9, *Assembly-Time Symbol Directives*, on page 4-21.

Example

This example shows the .float directive.

```
1 0000 E904        .float  -1.0e25
  0001 5951
2 0002 4040        .float  3
  0003 0000
3 0004 42F6        .float  123
  0005 0000
```

Syntax

.global *symbol*$_1$ [**,** ... **,** *symbol*$_n$]
.def *symbol*$_1$ [**,** ... **,** *symbol*$_n$]
.ref *symbol*$_1$ [**,** ... **,** *symbol*$_n$]

Description

The **.globa**l, **.def**, and **.ref** directives identify global symbols, which are defined externally or can be referenced externally.

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

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

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

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

A symbol may be declared global for two reasons:

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

Example

This example shows four files:

file1.lst and **file3.lst** 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. file1.lst uses the .global directive to identify these global symbols; file3.lst uses .ref and .def to identify the symbols.

file2.lst and **file4.lst** 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. 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 0000           INIT:
 6 0000 F000              ADD     #56h, A
   0001 0056
 7 0002 0000!             .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      0001      X:      .set    1
 6      0002      Y:      .set    2
 7      0003      Z:      .set    3
 8 0000 0000!             .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 0000           INIT:
 6 0000 F000              ADD     #56h, A
   0001 0056
 7 0002 0000!             .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      0001      X:      .set    1
 6      0002      Y:      .set    2
 7      0003      Z:      .set    3
 8 0000 0000!             .word   INIT
 9                ;       .
10                ;       .
11                ;       .
12                        .end
```

Syntax

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

Description

The following 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 blocks, 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). This 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 subsection 3.9.4, *Conditional Expressions*, on page 3-26.

Example This example shows conditional assembly.

```
 1                  SYM1    .set    1
 2                  SYM2    .set    2
 3                  SYM3    .set    3
 4                  SYM4    .set    4
 5
 6                  If_4:   .if     SYM4 = SYM2 * SYM2
 7 0000 0004                .byte   SYM4        ; Equal values
 8                          .else
 9                          .byte   SYM2 * SYM2 ; Unequal values
10                          .endif
11
12                  If_5:   .if     SYM1 <= 10
13 0001 000a                .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 0002 0008                .byte   SYM4 + SYM4  ; Equal values
22                          .endif
23
24                  If_7:   .if SYM1 = 2
25                          .byte   SYM1
26                          .elseif SYM2 + SYM3 = 5
27 0003 0005                .byte   SYM2 + SYM3
28                          .endif
```

Syntax

.int *value*$_1$ [**,** ... **,** *value*$_n$]
.**word** *value*$_1$ [**,** ... **,** *value*$_n$]

Description

The **.int** and **.word** directives are equivalent; they place one or more values into consecutive 16-bit fields in the current section.

The *values* can be either absolute or relocatable expressions. 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. If you use a label, it points to the first word that is initialized.

When you use .int 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 Section 4.9, *Assembly-Time Symbol Directives*, on page 4-21.

Example 1

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

```
1 0000              .space 73h
2 0000              .bss   PAGE, 128
3 0080              .bss   SYMPTR, 3
4 0008 E856  INST:  LD     #056h, A
5 0009 000A         .int   10, SYMPTR, –1, 35 + 'a', INST
  000a 0080–
  000b FFFF
  000c 0084
  000d 0008'
```

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 0000 0C80  WORDX: .word  3200, 1 + 'AB', –0AFh, 'X'
  0001 4143
  0002 FF51
  0003 0058
```

Syntax

.label *symbol*

Description

The **.label** directive defines a special *symbol* that refers to the loadtime address rather than the runtime 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 wish to load a block of performance-critical code into slower off-chip memory to save space, and then move the code to high-speed on-chip 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 runtime address so that references to the section (such as branches) are correct when the code runs.

The .label directive creates a special label that refers to the *loadtime* 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 loadtime address label.

```
      .sect  ".EXAMP"
      .label EXAMP_LOAD ; load address of section.
START:                  ; run address of section.
      <code>
FINISH:                 ; run address of section end.
      .label EXAMP_END  ; load address of section end.
```

For more information about assigning runtime and loadtime addresses in the linker, see Section 9.9, *Specifying a Section's Runtime Address*, on page 9-39.

Syntax

.length *page length*
.width *page width*

Description

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
- 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
- Minimum width: 80 characters
- Maximum width: 200 characters

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

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

Example

In this example, the page length and width are changed.

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

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

Syntax

```
.list
.nolist
```

Description

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

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

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

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

By default, the source listing is printed to the listing file; the assembler acts as if the .list directive had been specified.

Note: Creating a Listing File (–l Option)

If you don't request a listing file when you invoke the assembler, 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. Note also that 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 0000 0020           .word 32, 1 + ’A’
         0001 0042
       2              * Back in original file
       3 0002 F495           NOP
       7              * Back in original file
       8 0005 0044           .string ″Done″
         0006 006F
         0007 006E
         0008 0065
```

Syntax

.long *value*$_1$ [, ... , *value*$_n$]
.xlong *value*$_1$ [, ... , *value*$_n$]

Description

The **.long** and **.xlong** directives place one or more 32-bit values into consecutive words in the current section. The most significant word is stored first. The .long directive aligns the result on the long word boundary, while the .xlong directive does not.

The *value* operand can be either an absolute or 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 with labels.

You can use up to 100 values, but they must fit on a single source statement line. If you use a label, it points to the first word that is initialized.

When you use .long in a .struct/.endstruct sequence, .long defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, see Section 4.9, *Assembly-Time Symbol Directives*, on page 4-21.

Example

This example shows how the .long and .xlong directives initialize double words.

```
1 0000 0000  DAT1: .long  0ABCDh, 'A' + 100h, 'g', 'o'
  0001 ABCD
  0002 0000
  0003 0141
  0004 0000
  0005 0067
  0006 0000
  0007 006F
2 0008 0000'       .xlong  DAT1, 0AABBCCDDh
  0009 0000
  000a AABB
  000b CCDD
3 000c       DAT2:
```

Syntax

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

Description

Three directives enable 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 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 is optional, along with its expression. When the expression is false (0), the loop continues. When the expression is true (nonzero), or omitted, the assembler breaks the loop and assembles the code after the .endloop directive.

The **.endloop** directive terminates a repeatable block of code; it executes when the .break directive is true (nonzero) or when 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.

```
       1                    .eval       0,x
       2              COEF  .loop
       3                    .word       x*100
       4                    .eval       x+1, x
       5                    .break      x = 6
       6                    .endloop
1        0000 0000          .word       0*100
1                           .eval       0+1, x
1                           .break      1 = 6
1        0001 0064          .word       1*100
1                           .eval       1+1, x
1                           .break      2 = 6
1        0002 00C8          .word       2*100
1                           .eval       2+1, x
1                           .break      3 = 6
1        0003 012C          .word       3*100
1                           .eval       3+1, x
1                           .break      4 = 6
1        0004 0190          .word       4*100
1                           .eval       4+1, x
1                           .break      5 = 6
1        0005 01F4          .word       5*100
1                           .eval       5+1, x
1                           .break      6 = 6
```

Syntax

.mlib [**"**]*filename*[**"**]

Description

The **.mlib** directive provides the assembler with the name of a macro library. A macro library is a collection of files that contain macro definitions. These files are bound into a single file (called a library or archive) by the archiver. Each member of a macro library may contain one macro definition that corresponds to the name of the file. Macro library members must be *source* files (not object files).

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 may be enclosed in double quotes. You can specify a full pathname (for example, c:\dsp\macs.lib). 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 environment variable A_DIR

For more information about the –i option and the environment variable, see Section 3.4, *Naming Alternate Directories for Assembler Input*, on page 3-6.

When the assembler encounters a .mlib directive, it opens the library 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.

Example

This example creates a macro library that defines two macros, inc1 and zac1. The file inc1.asm contains the definition of inc1, and zac1.asm contains the definition of zac1.

inc1.asm	zac1.asm

`* Macro for incrementing`
`inc1  .macro`
`ADD #1,A`
`ADD #1,B`
`.endm`
	`* Macro for zero accumulators`
	`zac1  .macoro`
	`SUB A,A`
	`SUB B,B`
	`.endm`

Use the archiver to create a macro library:

```
ar500 –a mac inc1.asm zac1.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 0000           zac1             ; Macro call
1        0000 F420         SUB A,A
1        0001 F720         SUB B,B
       3 0002           inc1             ; Macro call
1        0002 F000         ADD #1,A
         0003 0001
1        0004 F300         ADD #1,B
         0005 0001
```

Syntax

```
.mlist
.mnolist
```

Description

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

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

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

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

Example

This example defines a macro named *STR_3*. The first time the macro is called, the macro expansion is not listed, because a .mnolist directive was assembled. The second time the macro is called, the macro expansion is listed, because a .mlist directive was assembled.

```
       1            STR_3  .macro   P1, P2, P3
       2                   .string ":p1:", ":p2:", ":p3:"
       3                   .endm
       4
       5 0000              STR_3 "as", "I", "am"
1        0000 003A          .string ":p1:", ":p2:", ":p3:"
         0001 0070
         0002 0031
         0003 003A
         0004 003A
         0005 0070
         0006 0032
         0007 003A
         0008 003A
         0009 0070
         000a 0033
         000b 003A
       6                   .mnolist
       7 000c              STR_3 "as", "I", "am"
       8                   .mlist
       9 0018              STR_3 "as", "I", "am"
1        0018 003A          .string ":p1:", ":p2:", ":p3:"
         0019 0070
         001a 0031
         001b 003A
         001c 003A
         001d 0070
         001e 0032
         001f 003A
         0020 003A
         0021 0070
         0022 0033
         0023 003A
```

Syntax

```
.mmregs
```

Description

The **.mmregs** directive defines global symbolic names for the ’C54x registers and places them in the global symbol table. It is equivalent to executing AL .set 8, AH .set 9, etc. The symbols are local and absolute. Using the .mmregs directive makes it unnecessary to define these symbols.

Table 4–9. Memory-Mapped Registers

Name	Hexadecimal-Address	Description
IMR	0000	Interrupt mask register
IFR	0001	Interrupt flag register
–	2–5	Reserved
ST0	0006	Status 0 register
ST1	0007	Status 1 register
AL	0008	A accumulator low (A [15:0])
AH	0009	A accumulator high (A [31:16])
AG	000A	A accumulator guard (A [39:32])
BL	000B	B accumulator low (B [15:0])
BH	000C	B accumulator high (B [31:16])
BG	000D	B accumulator guard (B [39:32])
T	000E	Temporary register
TRN	000F	Transition register
BK	0019	Circular size register
BRC	001A	Block repeat counter
RSA	001B	Block repeat start address
REA	001C	Block repeat end address
PMST	001D	PMST register
DRR0	0020	Data receive register 0
BDRR	0020	Data receive register

Note: Duplication of address values in the table supports the different names of the registers as they are implemented on different ’C54x devices.

Table 4–9. Memory-Mapped Registers (Continued)

Name	Hexadecimal-Address	Description
BDDR0	0020	BSP0 data receive register
DRR	0020	Data receive register
DXR0	0021	Data transmit register 0
BDXR	0021	Data transmit register
BDXR0	0021	Data transmit register
DXR	0021	Data transmit register
SPC0	0022	Serial port control register 0
BSPC	0022	Serial port control register
SPC	0022	Serial port control register
BSPCE	0023	BSP control extension register
BSPCE0	0023	BSP control extension register
SPCE	0023	BSP control extension register
TIM	0024	Timer register
PRD	0025	Period register
TCR	0026	Timer control register
PDWSR	0028	Program/data S/W wait-state register
SWWSR	0028	Program/data S/W wait-state register
IOWSR	0029	Bank-switching control register
BSCR	0029	Bank-switching control register
HPIC	002C	HPI control register
DRR1	0030	Data receive register 1
TRCV	0030	Data receive register
DXR1	0031	Data transmit register 1
TDXR	0031	Data transmit register

Note: Duplication of address values in the table supports the different names of the registers as they are implemented on different ’C54x devices.

Table 4–9. Memory-Mapped Registers (Continued)

Name	Hexadecimal-Address	Description
SPC1	0032	Serial port control register 1
TSPC	0032	Serial port control register
TRAD	0035	TDM receive address register
AXR	0038	ABU transmit address register
TCSR	0033	TDM channel select register
TRTA	0034	TDM receive/transmit register
AXR0	0038	ABU transmit address register
ARX	0038	ABU transmit address register
BKX	0039	ABU transmit buffer size register
BKX0	0039	ABU transmit buffer size register
ARR	003A	ABU receive address register
ARR0	003A	ABU receive address register
BKR	003B	ABU receive buffer size register
AXR1	003C	ABU transmit address register
BKX1	003D	ABU transmit buffer size register
ARR1	003E	ABU receive address register
BKR1	003F	ABU receive buffer size register
BDRR1	0040	BSP data receive register
BDXR1	0041	Data transmit register
BSPC1	0042	BSP control register
BSPCE1	0043	BSP control extension register
CLKMD	0058	Clock modes register
XPC	001E	Extended memory map register

Note: Duplication of address values in the table supports the different names of the registers as they are implemented on different ’C54x devices.

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. A local label, like other labels, points to an instruction word. Unlike other labels, local labels cannot be used in expressions. Local labels are not included in the symbol table.

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

Example

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

```
 1                      .ref   ADDRA, ADDRB, ADDRC
 2      0076  B         .set   76h
 3
 4 0000 1000! LABEL1:  LD     ADDRA, A
 5 0001 F010           SUB    #B, A
   0002 0076
 6 0003 F843           BC     $1, ALT
   0004 0008'
 7 0005 1000!          LD     ADDRB, A
 8 0006 F073           B      $2
   0007 0009'
 9
10 0008 1000! $1       LD     ADDRA, A
11 0009 0000! $2       ADD    ADDRC, A
12                     .newblock    ; Undefine $1 to reuse
13 000a F843           BC     $1, ALT
   000b 000D'
14 000c 8000!          STL    A, ADDRC
15 000d F495  $1       NOP
```

Syntax

```
.option  option list
```

Description

The **.option** directive selects several options for the assembler output listing. *Option list* is a list of options separated by vertical lines; each option selects a listing feature. These are valid options:

B	limits the listing of .byte directives to one line.
L	limits the listing of .long directives to one line.
M	turns off macro expansions in the listing.
R	resets the B, M, T, and W options.
T	limits the listing of .string directives to one line.
W	limits the listing of .word directives to one line.
X	produces a symbol cross-reference listing. (You can also obtain a cross-reference listing by invoking the assembler with the –x option.)

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, .word, **
 3                  ** .long, and .string directives to 1 **
 4                  **         to 1 line each.            **
 5                  ****************************************
 6                          .option B, W, L, T
 7 0000 00BD                .byte   –'C', 0B0h, 5
 8 0004 AABB                .long   0AABBCCDDh, 536 + 'A'
 9 0008 15AA                .word   5546, 78h
10 000a 0045                .string "Extended Registers"
11
12                  ****************************************
13                  **     Reset the listing options.     **
14                  ****************************************
15                          .option R
16 001c 00BD                .byte   –'C', 0B0h, 5
   001d 00B0
   001e 0005
17 0020 AABB                .long   0AABBCCDDh, 536 + 'A'
   0021 CCDD
   0022 0000
   0023 0259
18 0024 15AA                .word   5546, 78h
   0025 0078
19 0026 0045                .string "Extended Registers"
   0027 0078
   0028 0074
   0029 0065
   002a 006E
   002b 0064
   002c 0065
   002d 0064
   002e 0020
   002f 0052
   0030 0065
   0031 0067
   0032 0069
   0033 0073
   0034 0074
   0035 0065
   0036 0072
   0037 0073
```

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 it. 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:

```
TMS320C54x COFF Assembler       Version x.xx     Fri Aug 23 13:06:08 1996
 Copyright (c) 1993–1996    Texas Instruments Incorporated
**** Page Directive Example ****                                 PAGE    1

       2            ;        .
       3            ;        .
       4            ;        .
TMS320C54x COFF Assembler       Version x.xx     Fri Aug 23 13:06:08 1996
 Copyright (c) 1993–1996    Texas Instruments Incorporated
**** Page Directive Example ****                                 PAGE    2
```

Syntax

.sblock [”]*section name*[”] [**,** ”*section name*”, . . .]

Description

The **.sblock** directive designates sections for blocking. Blocking is an address alignment mechanism similar to page alignment, but weaker. A blocked section is guaranteed to not cross a page boundary (128 words) if it is smaller than a page, or to start on a page boundary if it is larger than a page. This directive allows specification of blocking for initialized sections only, not uninitialized sections declared with .usect or the .bss directives. The *section names* may optionally be enclosed in quotes.

Example

This example designates the .text and .data sections for blocking.

```
1 ****************************************
2 ** Specify blocking for the .text     **
3 ** and .data sections.                **
4 ****************************************
5        .sblock       .text, .data
```

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 begins assembling source code into the named section.

The *section name* identifies a section that the assembler assembles code into. The name can be up to 200 characters and must be enclosed in double quotes. A section name can contain a subsection name in the form *section name*:*subsection name*. For COFF1 formatted files, only the first 8 characters are significant.

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 0000                  .text
 5 0000 E878             LD      #78h, A ; Assembled into .text
 6 0001 F000             ADD     #36h, A ; Assembled into .text
   0002 0036
 7                ********************************************
 8                **  Begin assembling into Sym_Defs section. **
 9                ********************************************
10 0000                  .sect   "Sym_Defs"
11 0000 3D4C             .float  0.05    ; Assembled into Sym_Defs
   0001 CCCD
12 0002 00AA  X:         .word   0AAh    ; Assembled into Sym_Defs
13 0003 F000             ADD     #X, A   ; Assembled into Sym_Defs
   0004 0002+
14                ********************************************
15                **   Begin assembling into Vars section.    **
16                ********************************************
17 0000                  .sect   "Vars"
18      0010  WORD_LEN       .set    16
19      0020  DWORD_LEN      .set    WORD_LEN * 2
20      0008  BYTE_LEN       .set    WORD_LEN / 2
21                ********************************************
22                **   Resume assembling into .text section.  **
23                ********************************************
24 0003                  .text
25 0003 F000             ADD     #42h, A ; Assembled into .text
   0004 0042
26 0005 0003             .byte   3, 4    ; Assembled into .text
   0006 0004
27                ********************************************
28                **   Resume assembling into Vars section.   **
29                ********************************************
30 0000                  .sect   "Vars"
31 0000 000D             .field  13, WORD_LEN
32 0001 0A00             .field  0Ah, BYTE_LEN
33 0002 0000             .field  10q, DWORD_LEN
   0003 0008
34
```

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 *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 can be made externally visible with the .def or .global directive. 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   **
 3                **   and use it instead of the register.    **
 4                ********************************************
 5      0011      AUX_R1  .set    AR1
 6 0000 7711              STM     #56h, AUX_R1
   0001 0056
 7
 8                ********************************************
 9                **   Set symbol index to an integer expr.   **
10                **    and use it as an immediate operand.   **
11                ********************************************
12      0035      INDEX   .equ    100/2 +3
13 0002 F000              ADD     #INDEX, A
   0003 0035
14
15                ********************************************
16                ** Set symbol SYMTAB to a relocatable expr. **
17                **   and use it as a relocatable operand.   **
18                ********************************************
19 0004 000A      LABEL   .word   10
20      0005'     SYMTAB  .set    LABEL + 1
21
22                ********************************************
23                **   Set symbol NSYMS equal to the symbol   **
24                **   INDEX and use it as you would INDEX.   **
25                ********************************************
26      0035      NSYMS   .set    INDEX
27 0005 0035              .word   NSYMS
```

Syntax

.space *size in bits*
.bes *size in bits*

Description

The **.space** and **.bes** directives reserve *size* number of bits in the current section and fill them with 0s.

When you use a label with the .space directive, it points to the *first* word reserved. When you use a label with the .bes directive, it points to the *last* word reserved.

Example

This example shows how memory is reserved with the .space and .bes directives.

```
   1                ******************************************
   2                **  Begin assembling into .text section.   **
   3                ******************************************
   4 0000                  .text
   5
   6                ******************************************
   7                **  Reserve 0F0 bits (15 words in the      **
   8                **            .text section.               **
   9                ******************************************
  10 0000                  .space  0F0h
  11 000f 0100             .word   100h, 200h
     0010 0200
  12                ******************************************
  13                **  Begin assembling into .data section.   **
  14                ******************************************
  15 0000                  .data
  16 0000 0049             .string "In .data"
     0001 006E
     0002 0020
     0003 002E
     0004 0064
     0005 0061
     0006 0074
     0007 0061
  17                ******************************************
  18                **  Reserve 100 bits in the .data section; **
  19                **   RES_1 points to the first word that   **
  20                **         contains reserved bits.         **
  21                ******************************************
  22 0008           RES_1:  .space  100
  23 000f 000F             .word   15
  24 0010 0008"            .word   RES_1
  25
  26                ******************************************
  27                **  Reserve 20 bits in the .data section;  **
  28                **  RES_2 points to the last word that     **
  29                **         contains reserved bits.         **
  30                ******************************************
  31 0012           RES_2:  .bes    20
  32 0013 0036             .word   36h
  33 0014 0012"            .word   RES_2
```

Syntax

```
.sslist
.ssnolist
```

Description

Two directives enable 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 inhibited. 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.

(a) Mnemonic example

```
       1 000000           .bss    ADDRX, 1
       2 000001           .bss    ADDRY, 1
       3 000002           .bss    ADDRA, 1
       4 000003           .bss    ADDRB, 1
       5           ADD2   .macro  ADDRA, ADDRB
       6                  LD      ADDRA, A
       7                  ADD     ADDRB, A
       8                  STL     A, ADDRB
       9                  .endm
      10
      11 0000008094       STL     A, *AR4+
      12 000001           ADD2    ADDRX, ADDRY
1        0000011000-      LD      ADDRX, A
1        0000020001-      ADD     ADDRY, A
1        0000038001-      STL     A, ADDRY
      13
      14                  .sslist
      15
      16 000004 8094      STL     A, *AR4+
      17 000005 8090      STL     A, *AR0+
      18
      19 000006           ADD2    ADDRX, ADDRY
1        0000061000-      LD      ADDRA, A
#                         LD      ADDRX, A
1        0000070001-      ADD     ADDRB, A
#                         ADD     ADDRY, A
1        0000088001-      STL     A, ADDRB
#                         STL     A, ADDRY
```

(b) Algebraic example

```
       1 000000           .bss    ADDRX, 1
       2 000001           .bss    ADDRY, 1
       3 000002           .bss    ADDRA, 1
       4 000003           .bss    ADDRB, 1
       5           ADD2   .macro  ADDRA, ADDRB
       6                  A = ADDRA
       7                  A = A + ADDRB
       8                  ADDRB = A
       9                  .endm
      10
      11 0000008094       *AR4+ = A
      12 000001           ADD2    ADDRX, ADDRY
1        0000011000-      A = @ADDRX
1        0000020001-      A = A + ADDRY
1        0000038001-      @ADDRY = A
      13
      14                  .sslist
      15
      16 000004 8094      *AR4+ = A
      17 000005 8090      *AR0+ = A
      18
      19 000006           ADD2    ADDRX, ADDRY
1        0000061000-      A = ADDRA
#                         A = @ADDRX
1        0000070001-      A = A + ADDRB
#                         A = A + @ADDRY
1        0000088001-      ADDRB = A
#                         @ADDRY = A
```

Syntax

.string "*string*$_1$" [, ... , "*string*$_n$"]
.pstring "*string*$_1$" [, ... , "*string*$_n$"]

Description

The **.string** and **.pstring** directives place 8-bit characters from a character string into the current section. With the .string directive, each 8 bit character has its own 16-bit word, but with the .pstring directive, the data is packed so that each word contains two 8-bit bytes. Each *string* is either:

- An expression that the assembler evaluates and treats as a 16-bit signed number, or
- A character string enclosed in double quotes. Each character in a string represents a separate byte.

With .pstring, values are packed into words starting with the most significant byte of the word. Any unused space is padded with null bytes.

The assembler truncates any values that are greater than 8 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 word that is initialized.

Note that 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 Section 4.9, *Assembly-Time Symbol Directives*, on page 4-21.

Example

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

```
1 0000 0041  Str_Ptr:  .string  "ABCD"
  0001 0042
  0002 0043
  0003 0044
2 0004 0041            .string  41h, 42h, 43h, 44h
  0005 0042
  0006 0043
  0007 0044
3 0008 4175            .pstring "Austin", "Houston"
  0009 7374
  000a 696E
  000b 486F
  000c 7573
  000d 746F
  000e 6E00
4 000f 0030            .string  36 + 12
```

Syntax

```
[ stag ]      .struct          [ expr ]
[ mem0 ]      element          [ expr0 ]
[ mem1 ]      element          [ expr1 ]
      .           .                .
      .           .                .
      .           .                .
[ memn ]      .tag stag        [ exprn]
      .           .                .
      .           .                .
      .           .                .
[ memN ]      element          [ exprN ]
[ size ]      .endstruct
 label        .tag             stag
```

Description

The **.struct** directive assigns symbolic offsets to the elements of a data structure definition. This enables you to group similar data elements together and then let the assembler calculate the element offset. This is similar to a C structure or a Pascal record.

> **Note:**
>
> The .struct directive does not allocate memory. It merely creates a symbolic template that can be used repeatedly.

The **.endstruct** directives 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.

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. Stag is optional for .struct, but required for .tag.

expr is an optional expression indicating the beginning offset of the structure. Structures default to start at 0.

mem_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.

element is one of the following descriptors: .string, .byte, .word, .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 a stag must be used (as in the definition).

$expr_n$ is an optional expression for the number of elements described. This value defaults to 1. A .string element is considered to be one word in size, and a .field element is one bit.

size is an optional label for the total size of the structure.

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          0000  NOM        .int                     ; member1 = 0
 3          0001  DEN        .int                     ; member2 = 1
 4          0002  REAL_LEN   .endstruct               ; real_len = 4
 5
 6 0000 0001-                ADD  REAL + REAL_REC.DEN, A
 7                                                    ; access structure element
 8
 9 0000                      .bss REAL, REAL_LEN      ; allocate mem rec
```

Example 2

```
 1                CPLX_REC   .struct
 2          0000  REALI      .tag REAL_REC            ; stag
 3          0002  IMAGI      .tag REAL_REC            ; member1 = 0
 4          0004  CPLX_LEN   .endstruct               ; cplx_len = 4
 5
 6                COMPLEX    .tag CPLX_REC            ; assign structure attrib
 7
 8 0002                      .bss COMPLEX, CPLX_LEN
 9
10 0001 0002-                ADD  COMPLEX.REALI, A    ; access structure
11 0002 8002-                STL  A, COMPLEX.REALI
12
13 0003 0104-                ADD  COMPLEX.IMAGI, B    ; allocate space
```

Example 3

```
 1                           .struct                  ; no stag puts mems into
 2                                                    ; global symbol table
 3          0000  X          .int                     ; create 3 dim templates
 4          0001  Y          .int
 5          0002  Z          .int
 6          0003             .endstruct
```

Example 4

```
 1                BIT_REC    .struct                  ; stag
 2          0000  STREAM     .string 64
 3          0040  BIT7       .field  7                ; bits1 = 64
 4          0040  BIT9       .field  9                ; bits2 = 64
 5          0041  BIT10      .field  10               ; bits3 = 65
 6          0042  X_INT      .int                     ; x_int = 67
 7          0043  BIT_LEN    .endstruct               ; length = 68
 8
 9                BITS       .tag BIT_REC
10 0000 0040-                ADD  BITS.BIT7, A        ; move into acc
11 0001 f030                 AND  #007Fh, A           ; mask off garbage bits
   0002 007f
12
13 0000                      .bss BITS, BIT_REC
```

Syntax

```
.tab  size
```

Description

The **.tab** directive defines the tab size. Tabs encountered in the source input are translated to *size* spaces in the listing. The default tab size is eight spaces.

Example

Each of the following lines 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 0000 F495          NOP
       3 0001 F495          NOP
       4 0002 F495          NOP
       5
       7 0003 F495       NOP
       8 0004 F495       NOP
       9 0005 F495       NOP
      10
      12 0006 F495                    NOP
      13 0007 F495                    NOP
      14 0008 F495                    NOP
```

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 specify a different sections directives (.data or .sect).

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. The .data section contains integer constants, and the .text section contains character strings.

```
 1                ****************************************
 2                ** Begin assembling into .data section.**
 3                ****************************************
 4 0000                  .data
 5 0000 000a             .byte   0Ah, 0Bh
   0001 000b
 6
 7                ****************************************
 8                ** Begin assembling into .text section. **
 9                ****************************************
10 0000                  .text
11 0000 0041  START:    .string "A","B","C"
   0001 0042
   0002 0043
12 0003 0058  END:      .string "X","Y","Z"
   0004 0059
   0005 005a
13
14 0006 0000'           ADD     START, A
15 0007 0003'           ADD     END, A
16               ****************************************
17               **  Resume assembling into .data section.**
18               ****************************************
19 0002                 .data
20 0002 000c            .byte   0Ch, 0Dh
   0003 000d
21               ****************************************
22               **  Resume assembling into .text section.**
23               ****************************************
24 0008                 .text
25 0008 0051            .string "Quit"
   0009 0075
   000a 0069
   000b 0074
```

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 65 characters. If you supply more than 65 characters, the assembler truncates the string and issues a warning.

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 on succeeding pages.

Source file:

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

Listing file:

```
TMS320C54x COFF Assembler       Version x.xx     Fri Aug 23 16:25:49 1996
 Copyright (c) 1993-1996    Texas Instruments Incorporated

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

       2            ;        .
       3            ;        .
       4            ;        .
TMS320C54x COFF Assembler       Version x.xx     Fri Aug 23 16:25:49 1996
 Copyright (c) 1993-1996    Texas Instruments Incorporated

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

Syntax

symbol **.usect "***section name***"***, size in words* [**,** *blocking flag*] [**,** *alignment flag*]

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 have no contents. However, .usect defines additional sections that can be placed anywhere in memory, independently of the .bss section.

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're reserving space.

section name must be enclosed in double quotes. This parameter names the uninitialized section. The name can be up to 200 characters. For COFF1 formatted files, only the first 8 characters are significant. A section name can contain a subsection name in the form *section name*:*subsection name*.

size in words is an expression that defines the number of words that are reserved in section *name.*

blocking flag is an optional parameter. If specified and nonzero, the flag means that this section will be blocked. Blocking is an address mechanism similar to alignment, but weaker. It means a section is guaranteed to not cross a page boundary (128 words) if it is smaller than a page, and to start on a page boundary if it is larger than a page. This blocking applies to the section, not to the object declared with this instance of the .usect directive.

alignment flag is an optional parameter. This flag causes the assembler to allocate size on long word boundaries.

Other sections directives (.text, .data, and .sect) end the current section and tell the assembler to begin assembling into another section. The .usect and the .bss directives, however, do not affect the current section. The assembler assembles the .usect and the .bss directives and then resumes assembling into 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.

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

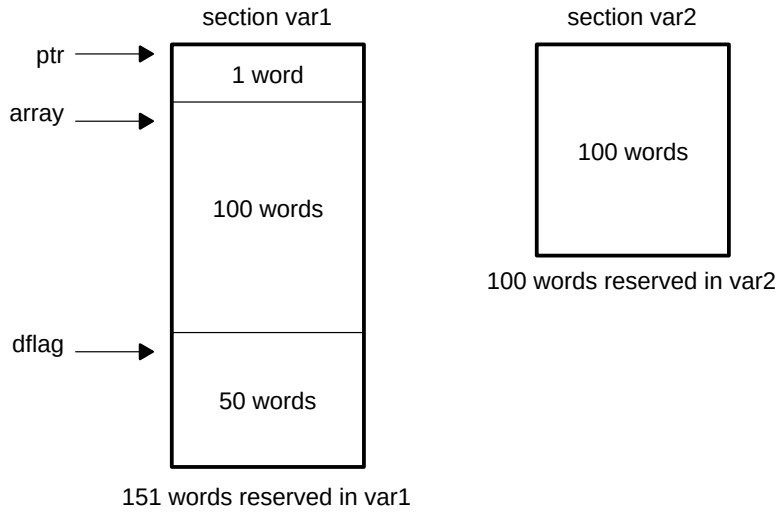
Example

This example uses the .usect directive to define two uninitialized, named sections, var1 and var2. The symbol ptr points to the first word reserved in the var1 section. The symbol array points to the first word in a block of 100 words reserved in var1, and dflag points to the first word in a block of 50 words in var1. The symbol vec points to the first word reserved in the var2 section.

Figure 4–7 on page 4-85 shows how this example reserves space in two uninitialized sections, var1 and var2.

```
 1                ****************************************
 2                **     Assemble into .text section.     **
 3                ****************************************
 4 0000                    .text
 5 0000 E803               LD      #03h, A
 6
 7                ****************************************
 8                **       Reserve 1 word in var1.        **
 9                ****************************************
10 0000           ptr     .usect  "var1", 1
11
12                ****************************************
13                **      Reserve 100 words in var1.      **
14                ****************************************
15 0001           array   .usect  "var1", 100
16
17 0001 F000              ADD     #037h, A  ; Still in .text
   0002 0037
18
19                ****************************************
20                **      Reserve 50 words in var1.       **
21                ****************************************
22 0065           dflag   .usect  "var1", 50
23
24 0003 F000              ADD     #dflag, A ; Still in .text
   0004 0065-
25
26                ****************************************
27                **      Reserve 100 words in var2.      **
28                ****************************************
29 0000           vec     .usect  "var2", 100
30
31 0005 F000              ADD     #vec, A   ; Still in .text
   0006 0000-
32                ****************************************
33                **  Declare an external .usect symbol.  **
34                ****************************************
35                        .global array
```

Figure 4–7. The .usect Directive



Syntax

```
.version  value
```

Description

The **.version** directive determines for which processor instructions are built. Use one of the following for *value*:

541
542
543
545
545LP
546LP
548

Chapter 5

Mnemonic Instruction Set Summary

The TMS320C54x device supports general-purpose instructions as well as arithmetic-intensive instructions that are particularly suited for DSP and other numeric-intensive applications. There are two instruction sets—mnemonic and algebraic. These two sets perform the same functions but with very different syntax.

This chapter contains a summary of the mnemonic instruction set. Table entries show the syntax for the instruction and describes the instruction operation. Section 5.1, *Using the Summary Tables*, shows a sample table entry and describes the abbreviations used in the table. Section 5.2, *Mnemonic and Algebraic Instruction Sets Cross-Reference,* on page 5-5 cross references the mnemonic instruction set to the algebraic instruction set.

This chapter does not cover topics such as opcodes, instruction timing, or addressing modes.

Topic **Page**

5.1 Using the Summary Tables

To help you read the summary table, this section provides an example of a table entry and lists the acronyms used in the table.

5.1.1 Table Entry Example

This is how the AND mnemonic instruction appears in the summary table:

Example 5–1. Table Entry for a Mnemonic Instruction

Syntax	Description
AND *Smem*, *src* **AND** *#lk*, [*SHIFT1*,] *src* [, *dst*] **AND** *#lk*, *16*, *src* [, *dst*] **AND** *src* [, *SHIFT*] [,*dst*]	**AND With Accumulator** AND the data value or constant with the source accumulator (A or B). If a shift is specified, left shift the data value before the AND. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.

5.1.2 Table Entry Explained

The syntax column lists the syntax for the AND instruction. Alternative syntax is shown in the lines following the first syntax. Abbreviations used in syntax are found in subsection 5.1.3, *Symbols and Acronyms*, on page 5-2.

The description column briefly describes how the instruction functions. Often, an instruction functions slightly differently with different parameters. For complete information about an instruction, see the *TMS320C54x User's Guide*.

5.1.3 Symbols and Acronyms

The following table lists the instruction set symbols and acronyms used throughout this chapter.

Table 5–1. Symbols and Acronyms Used in the Instruction Set Summary

Symbol	Definition
( )	Contents of
[ ]	Optional items; the brackets are not part of the syntax
#	Prefix of constants used in immediate addressing
| |	Parallel instructions
ARP	Auxiliary register pointer

Table 5–1. Symbols and Acronyms Used in the Instruction Set Summary (Continued)

Symbol	Definition
ARX	Auxiliary register (AR0–AR7)
ASM	Accumulator shift mode (from ST1)
Borrow	A literal that specifies to subtract with borrow
BRC	Block repeat counter
Carry	Carry bit
C16	Dual 16-bit/double-precision bit
CC	Condition code
CMPT	Compatibility mode bit
cond	Conditional expression
data	Data memory access which can be incremented under a single repeat instruction
dbl	32-bit access
dmad	16-bit data immediate addressed value
DP	Data page pointer
dst	Destination accumulator (A or B)
$\overline{\text{dst}}$	Accumulator opposite the previous dst
dst1, dst2	If *dst1* = A, then *dst2* = B If *dst1* = B, then *dst2* = A
dual	Dual access
extpmad	23-bit immediate program-memory address
HI	High half of the register
INTM	Interrupt mask bit
K	Short immediate value (less than 9 bits)
k3	3-bit immediate value ($0 \leq k3 \leq 7$)
k5	5-bit immediate value ($-16 \leq k5 \leq 15$)
k9	9-bit immediate value ($0 \leq k9 \leq 511$)
lk	Long immediate value (16 bits)
Lmem	32-bit single-addressed mode value (direct or indirect) (long word addressing)
mmr, MMR	Memory-mapped register

Table 5–1. Symbols and Acronyms Used in the Instruction Set Summary (Continued)

Symbol	Definition
MMRx, MMRy	Memory-mapped register, AR0–AR7 or SP
n	Operand for the XC instruction indicating the number of instructions to conditionally execute ❑ 1 = One instruction executes ❑ 0 = Two instructions execute
N	Status register for RSBX/SSBX instruction; either 0 or 1
OVM	Overflow mode bit
PA	16-bit port immediate addressed value
PC	Program counter
pmad	16-bit program immediate addressed value
port	I/O port access
prog	Program memory access
RC	Repeat counter
rnd	Round the expression
SBIT	Status register bit for RSBX/SSBX instruction; $0 \leq$ SBIT $\leq$ 15
SHIFT	Shift value in –16 to +15 range
SHIFT1	Shift value in 0 to 15 range
SHIFT2	Shift value in 0 to 16 range
Sind	Single indirect address mode
Smem	16-bit single-addressed mode value (direct or indirect)
SP	Stack pointer register
src	Source accumulator (A or B)
T	A literal that specifies to use TREG as operand
TC	Test/control flag bit
TREG	Temporary register
TRN	Transition register
TS	Shift value held in TREG (–16, +31 range)
uns	Unsigned operand
Xmem	16-bit dual addressed mode value (indirect only) used in dual operand instructions and some single-operand instructions
Ymem	16-bit dual addressed mode value (indirect only) used mainly in dual-operand instructions

5.2 Mnemonic and Algebraic Instruction Sets Cross-Reference

The following table cross references the mnemonic instruction set with the algebraic instruction set. A summary of an instruction is shown on the indicated page. See the *TMS320C54x User's Guide* for detailed information about the instruction sets.

To translate mnemonic code to algebraic code use the translator utility (see Chapter 13, *Mnemonic to Algebraic Translator Description*).

Mnemonic Instruction	Page	Equivalent Algebraic Instruction	Page
ABDST *Xmem*, *Ymem*	5-14	abdst (*Xmem, Ymem*)	6-27
ABS *src* [,*dst*]	5-14	*dst* = |*src*|	6-27
ADD *Smem, src* ADD *Smem, TS, src* ADD *Smem,* 16, *src* [,*dst*] ADD *Smem* [,*SHIFT*], *src* [,*dst*] ADD *Xmem, SHIFT1*, *src* ADD *Xmem, Ymem, dst* ADD #*lk* [,*SHIFT2*], *src* [,*dst*] ADD #*lk*, 16, *src* [,*dst*] ADD *src* [,*SHIFT*] [,*dst*] ADD *src, ASM* [,*dst*]	5-14	*src* = *src* + *Smem* *src* += *Smem* *src* = *src* + *Smem* << *TS* *src* += *Smem* << *TS* *dst* = *src* + *Smem* << 16 *dst* += *Smem* << 16 *dst* = *src* + *Smem* [<< *SHIFT*] *dst* += *Smem* [<< *SHIFT*] *src* = *src* + *Xmem* << *SHIFT1* *src* += *Xmem* << *SHIFT1* *dst* = *Xmem* << *16* + *Ymem* << 16 *dst* = *src* + #*lk* [<< *SHIFT1*] *dst* += #*lk* [<< *SHIFT1*] *dst* = *src* + #*lk* << 16 *dst* += #*lk* << 16 *dst* = *dst* + *src* [<< *SHIFT*] *dst* += *src* [<< *SHIFT*] *dst* = *dst* + *src* << *ASM* *dst* += *src* << *ASM*	6-20
ADDC *Smem, src*	5-14	*src* = *src* + *Smem* + *Carry* *rc* += *Smem* + *Carry*	6-20
ADDM *#lk*, *Smem*	5-14	*Smem* = *Smem* + *#lk* *Smem* += *#lk*	6-20
ADDS *Smem, src*	5-14	*src* = *src* + uns*(Smem)* *src* += uns*(Smem)*	6-20

Mnemonic Instruction	Page	Equivalent Algebraic Instruction	Page
AND *Smem*, *src* AND *#lk* [, *SHFT]*, *src* [,*dst*] AND *#lk*, 16, *src* [,*dst*] AND *src* [,*SHIFT*] [,*dst*]	5-14	*src = src* & *Smems* *src* &= *Smem* *dst = src & #lk* [<< *SHIFT1*] *dst* &= *#lk* [<< *SHIFT1*] *dst = src* & *#lk* << 16 *dst* &= *#lk* << 16 *dst = dst* & *src* [<< *SHIFT*] *dst* &= *src* [<< *SHIFT*]	6-28
ANDM *#lk*, *Smem*	5-14	*Smem = Sme*m & *#lk* *Smem* & = *#lk*	6-28
B[D] *pmad*	5-14	[d]goto *pmad*	6-32
BACC[D] *src*	5-15	[d]goto *src*	6-32
BANZ[D] *pmad*, *Sind*	5-15	if (*Sind* != *0)* [d]goto *pmad*	6-32
BC[D] *pmad*, *cond* [,*cond* [,*cond*]]	5-15	*if (cond* [,*cond* [,*cond*]]) [d]goto *pmad*	6-32
BIT *Xmem*, *bit_code*	5-15	*TC =* bit *(Xmem, bit_code)*	6-31
BITF *Smem*, *#lk*	5-15	*TC* = bitf (*Smem*, *#lk)*	6-31
BITT *Smem*	5-15	*TC* = bitt (*Smem)*	6-31
CALA[D] *src*	5-15	[d]call *src*	6-33
CALL[D] *pmad*	5-15	[d]call *extpmad*	6-33
CC[D] *pmad*, *cond* [,*cond* [,*cond*]]	5-15	if (*cond* [,*cond* [,*cond*]]*)* [d]call *pmad*	6-33
CMPL *src* [,*dst*]	5-15	*dst = ~src*	6-27
CMPM *Smem, lk*	5-15	*TC* = (*Smem* == *#lk*)	6-31
CMPR *CC*, AR*x*	5-16	*TC* = (AR0 == AR*x*) (==, <, >, !=) *TC* = (AR0 > AR*x*) *TC* = (AR0 < AR*x*) *TC* = (AR0 != AR*x*)	6-31
CMPS *src, Smem*	5-16	*cmps(src,Smem)*	6-39
DADD *Lmem,* src [,*dst*]	5-16	*dst = src* + dbl *(Lmem)* *dst* += dbl *(Lmem)* *dst = src* + dual *(Lmem)* *dst* += dual *(Lmem)*	6-25
DADST *Lmem*, *dst*	5-16	*dst* = dadst *(Lmem, T)*	6-25
DELAY *Smem*	5-16	delay *(Smem*)	6-27

Mnemonic Instruction	Page	Equivalent Algebraic Instruction		Page
DLD *Lmem*, *dst*	5-16	*dst* = dbl *(Lmem)* *dst* = dual *(Lmem)*		6-25
DRSUB *Lmem*, *src*	5-16	*src* = dbl *(Lmem) – src* *src* = dual *(Lmem) – src*		6-25
DSADT *Lmem*, *dst*	5-17	*dst* = dsadt *(Lmem, T)*		6-25
DST *src*, *Lmem*	5-17	dbl *(Lmem) = src* dual *(Lmem) = src*		6-25
DSUB *Lmem*, *src*	5-17	*src = src* – dbl *(mem)* *src* –= dbl *(Lmem)* *src = src* – dual *(mem)* *src* –= dual *(Lmem)*		6-25
DSUBT *Lmem*, *dst*	5-17	*dst* = dbl *(Lmem) – T* *dst* = dual *(Lmem) – T*		6-25
EXP *src*	5-17	*T* = exp *(src)*		6-27
FB[D] *extpmad*	5-18	far [d]goto *extpmad*		6-32
FBACC[D] *src*	5-18	far [d]goto *src*		6-32
FCALA[D] *src*	5-18	far [d]call *src*		6-33
FCALL[D] *extpmad*	5-18	far [d]call *extpmad*		6-33
FIRS *Xmem*, *Ymem*, *pmad*	5-18	firs *(Xmem, Ymem, pmad)*		6-27
FRAME *K*	5-18	*SP = SP + K* *SP += K*	(–128 <= K <= 127) (–128 <= K <= 127)	6-35
FRET[D]	5-18	far [d]return		6-34
FRETE[D]	5-18	far [d]return_enable		6-34
IDLE *K*	5-18	idle *(K)*	(0 <= K <= 3)	6-36
INTR *K*	5-18	int *(K)*	(0 <= K <= 31)	6-33
LD *Smem*, *dst* LD *Smem*, *TS*, *dst* LD *Smem, 16, dst* LD *Smem* [,*SHIFT*], *dst* LD *Xmem*, *SHIFT1*, *dst* LD *#K, dst* LD #*lk* [,*SHIFT1*], *dst* LD #*lk*, 16, *dst* LD *src*, *ASM* [,*dst*] LD *src* [,*SHIFT*] ,*dst*	5-19	*dst = Smem* *dst = Smem << TS* *dst = Smem <<* 16 *dst = Smem* [*<< SHIFT*] *dst = Xmem* [*<< SHIFT1*] *dst = #K* *dst = #lk* [*<< SHIFT1*] *dst = #lk <<* 16 *dst = src << ASM* *dst = src* [*<< SHIFT*]	 (0 <= K <= 255) 	6-37

Mnemonic Instruction	Page	Equivalent Algebraic Instruction	Page
LD *Smem, T*	5-19	*T = Smem*	6-37
LD *Smem, DP*		*DP = Smem*	
LD *#k9, DP*		*DP = #k9* (0 <= Kk9<= 511)	
LD *#k5, ASM*		*ASM = #k5* (–16 <= Kk5<= 15)	
LD *#k3, ARP*		*ARP = #k3* (0 <=*k3*K <= 7)	
LD *Smem, ASM*		*ASM = Smem* (ASM = Smem[0:4])	
LDM *MMR, dst*	5-19	*dst = MMR*	6-37
		dst = mmr(MMR)	
LD *Xmem*, *dst* || MAC *Ymem* [, *dst*]	5-19	*dst = Xmem* [<< *16*] || *dst* += *T*Ymem)*	6-40
LD *Xmem*, *dst* || MACR *Ymem* [, *dst*]		*dst = Xmem* [<< *16*] || *dst = dst + T*Ymem)* *dst = Xmem* [<< *16*] || *dst* = rnd *(dst + T*Ymem)*	
LD *Xmem*, *dst* || MAS *Ymem* [,*dst*]	5-19	*dst = Xmem* [<< *16*] *|| dst –= T * Ymem* *dst = Xmem* [<< *16*] *|| dst = dst – T * Ymem* *dst = Xmem* [<< *16*] *|| dst* = rnd (*dst – T * Ymem*)	6-40
LD *Xmem*, *dst* || MASR *Ymem* [, *dst*]			
LDR *Smem*, *dst*	5-19	*dst* = rnd *(Smem)*	6-37
LDU *Smem*, *dst*	5-19	*dst* = uns *(Smem)*	6-37
LMS *Xmem*, *Ymem*	5-19	lms *(Xmem, Ymem)*	6-27
LTD *Smem*	5-20	ltd *(Smem)*	6-37
MAC *Smem*, *src* MAC *Xmem*, *Ymem*, *src* [,*dst*] MAC *#lk*, *src* [,*dst*] MAC *Smem*, *#lk*, *src* [,*dst*] MACR *Smem*, *src* MACR *Xmem*, *Ymem*, *src* [,*dst*]	5-20	*dst = (src + T * Smem*) *dst += T * Smem* *dst = (src + Xmem * Ymem*) [, *T = Xmem*] *dst += Xmem * Ymem* [, *T = Xmem*] *dst = src + T * #lk* *dst += T * #lk* *dst = src + Smem * #lk* [, *T = Smem*] *dst += Smem * #lk* [, *T = Smem*] *dst* = rnd *(src + T * Smem)* *dst* = rnd *(src + Xmem * Ymem)* [, *T = Xmem*]	6-23
MACA *Smem* [, *B*] MACA T, *src* [,*dst*] MACAR *Smem* [, *B*] MACAR T, *src* [,*dst*]	5-20	*B = B + Smem* * HI *(A)* [, T = *Smem*] *B += Smem* * HI *(A)* [, T = *Smem*] *dst = src* + T * HI *(A)* *dst* += T * HI *(A)* *B* = rnd *(B + Smem* * HI *(A))* [, T = *Smem*] *dst* = rnd *(src* + T * HI *(A))*	6-23

Mnemonic Instruction	Page	Equivalent Algebraic Instruction	Page
MACD *Smem*, *pmad*, *src*	5-20	macd *(Smem, pmad, src)*	6-23
MACP *Smem*, *pmad*, *src*	5-20	macp *(Smem, pmad, src)*	6-23
MACSU *Xmem*, *Ymem*, *src*	5-20	*src* = *src* + uns *(Xmem) * Ymem* [, T = *Xmem*] *src* += uns *(Xmem) * Ymem* [, T = *Xmem*]	6-23
MAR *Smem* MAR *AR*n*+0 MAR *AR*n*–0	5-21	mar *(Smem)* AR*n* = AR*n* + AR0 AR*n* += AR0 AR*n* = AR*n* – AR0 AR*n* –= AR0	6-36
MAS *Smem, src* MAS *Xmem*, *Ymem*, *src* [,*dst*] MASR *Smem, src* MASR *Xmem*, *Ymem*, *src* [,*dst*]	5-21	*src* = *src* – T * *Smem* *src* –= T * *Smem* *dst* = *src* – *Xmem * Ymem* [, T = *Xmem*] *dst* –= *Xmem * Ymem* [, T = *Xmem*] *src* = rnd *(src* – T * *Smem)* *dst* = rnd *(src – Xmem * Ymem)* [, T=*Xmem*]	6-23
MASA *Smem* [, *B*] MASA T, *src* [, *dst*] MASAR T, *src* [, *dst*]	5-21	*B* = *B* – *Smem* * HI *(A)* [, T = *Smem*] *B* –= *Smem* * HI *(A)* [, T = *Smem*] *dst* = *src* – T * HI *(A)* *dst* –= T * HI *(A)* *dst* = rnd *(src* – T * HI *(A))*	6-23
MAX *dst*	5-21	*dst* = max *(a, b)*	6-27
MIN *dst*	5-21	*dst* = min *(a, b)*	6-27
MPY *Smem*, *dst* MPY *Xmem*, *Ymem*, *dst* MPY *Smem*, *#lk*, *dst* MPY *#lk*, *dst* MPYR *Smem*, *dst*	5-21	*dst* = T * *Smem* *dst* = *Xmem * Ymem [,* T = *Xmem]* *dst* = *Smem * #lk [,* T = *Smem]* *dst* = T * *#lk* *dst* = rnd *(*T * *Smem)*	6-22
MPYA *dst* MPYA *Smem*	5-22	*dst* = T * HI *(A)* *B* = *Smem* * HI *(A)* [, T = *Smem*]	6-22
MPYU *Smem*, *dst*	5-22	*dst* = T * uns *(Smem)*	6-22
MVDD *Xmem*, *Ymem*	5-22	*Ymem* = *Xmem*	6-42
MVDK *Smem*, *dmad*	5-22	data *(dmad)* = *Smem*	6-42
MVDM *dmad*, *MMR*	5-22	*MMR* = data *(dmad)* *mmr (MMR)* = data *(dmad)*	6-42
MVDP *Smem*, *pmad*	5-22	prog *(pmad)* = *Smem*	6-42
MVKD *dmad*, *Smem*	5-22	*Smem* = data *(dmad)*	6-42

Mnemonic Instruction	Page	Equivalent Algebraic Instruction	Page
MVMD *MMR*, *dmad*	5-22	data *(dmad) = MMR* data *(dmad) = mmr(MMR)*	6-42
MVMM *MMRx*, *MMRy*	5-22	*MMRy = MMRx* (0 <= x,y <= 8) *mmr(MMRy) = mmr(MMRx)*	6-42
MVPD *pmad*, *Smem*	5-22	*Smem* = prog *(pmad)*	6-42
NEG *src* [,*dst*]	5-23	*dst = –src*	6-27
NOP	5-23	nop	6-36
NORM *src* [,*dst*]	5-23	*dst = src << TS* *dst* = norm *(src,*T*S)*	6-27
OR *Smem, src* OR #*lk* [,*SHIFT2*], *src* [,*dst*] OR #*lk*, 16, *src* [,*dst*] OR *src* [,*SHIFT*] [,*dst*]	5-23	*src = src* | *Smem* *src* |= *Smem* *dst = src* | *#lk* [*<< SHIFT*] *dst* |= *#lk* [*<< SHIFT]* *dst = src* | *#lk << 16* *dst* |= *#lk << 16* *dst = dst* | *src* [*<< SHIFT*] *ds*t |= *src* [*<< SHIFT*]	6-29
ORM *#lk, Smem*	5-23	*Smem = Smem* | *#lk* *Smem* |= *#lk*	6-29
POLY *Smem*	5-23	poly *(Smem)*	6-27
POPD *Smem*	5-23	*Smem* = pop()	6-35
POPM *MMR*	5-23	*MMR* = pop() *mmr*(*MMR*) = pop()	6-35
PORTR *PA*, *Smem*	5-23	*Smem* = port *(PA)*	6-42
PORTW *Smem, PA*	5-24	port *(PA) = Smem*	6-42
PSHD *Smem*	5-24	push *(Smem)*	6-35
PSHM *MMR*	5-24	push *(MMR)* push *(mmr(MMR))*	6-35
RC[D] *cond* [,*cond* [,*cond*]]	5-24	if *(cond* [,*cond*] [,*cond*]*)* [d]return	6-34
READA *Smem*	5-24	*Smem* = prog *(A)*	6-42
RESET	5-24	reset	6-36
RET[D]	5-24	[d]return	6-34
RETE[D]	5-24	[d]return_enable	6-34

Mnemonic Instruction	Page	Equivalent Algebraic Instruction	Page
RETF[D]	5-24	[d]return_fast	6-34
RND *src* [,*dst*]	5-24	*dst* = rnd (*src*)	6-27
ROL *src*	5-25	*src = src* || *Carry*	6-30
ROLTC *src*	5-25	roltc *(src)*	6-30
ROR *src*	5-25	*src = src // Carry*	6-30
RPT *Smem* RPT #*K* RPT #*lk*	5-25	repeat *(Smem)* repeat *(#k)* (RC=#k) (0<=k<=255) repeat *(#lk)* (RC=#lk) (0<=lk<=65535)	6-35
RPTB *pmad* RPTBD *pmad*	5-25	[d]blockrepeat *(pmad)*	6-35
RPTZ *dst, #lk*	5-25	repeat *(#lk), dst = 0*	6-35
RSBX *N*, *SBIT*	5-25	*SBIT = 0* *ST(N,SBIT)=0*	6-36
SACCD *src, Xmem, cond*	5-25	if *(cond) Xmem* = HI *(src) << ASM*	6-39
SAT *src*	5-25	saturate *(src)*	6-27
SFTA *src*, *SHIFT* [,*dst*]	5-25	*dst = src << C SHIFT*	6-30
SFTC *src*	5-25	shiftc *(src)*	6-30
SFTL *src*, *SHIFT* [,*dst*]	5-26	*dst = src << SHIFT*	6-30
SQDST *Xmem*, *Ymem*	5-26	sqdst *(Xmem, Ymem)*	6-27
SQUR *Smem, dst* SQUR *A, dst*	5-26	*dst = Smem * Smem* [, T = *Smem*] *dst* = square *(Smem)* [, T = *Smem*] *dst =* HI *(A)* * HI *(A)* *dst =* square *(*HI *(A))*	6-22
SQURA *Smem, src*	5-26	*src = src* + square *(Smem)* [, T=*Smem*] *src* += square *(Smem)* [, T = *Smem*] *src = src* + *Smem * Smem* [, T=*Smem*] *src* += *Smem * Smem* [, T = *Smem*]	6-23
SQURS *Smem, src*	5-26	*src = src* – square *(Smem)* [, T=*Smem*] *src* –= square *(Smem)* [, T = *Smem*] *src = src – Smem * Smem* [, T=*Smem*] *src* –= *Smem * Smem* [, T = *Smem*]	6-23
SRCCD *Xmem*, *cond*	5-26	if *(cond) Xmem = BRC*	6-39

Mnemonic Instruction	Page	Equivalent Algebraic Instruction	Page
SSBX *N*, *SBIT*	5-26	*SBIT = 1* ST *(N,SBIT)=1*	6-36
ST T, *Smem* ST *TRN*, *Smem* ST *#lk*, *Smem*	5-26	*Smem =* T *Smem = TRN* *Smem = #lk*	6-38
STH *src*, *Smem* STH *src*, *ASM*, *Smem* STH *src*, *SHIFT1*, *Xmem* STH *src* [,*SHIFT*], *Smem*	5-26	*Smem =* HI *(src)* *Smem =* HI *(src) << ASM* *Xmem =* HI *(src) << SHIFT1* *Smem =* HI *(src) << SHIFT*	6-38
STL *src*, *Smem* STL *src*, *ASM*, *Smem* STL *src*, *SHIFT1*, *Xmem* STL *src* [,*SHIFT*], *Smem*	5-26	*Smem = src* *Smem = src << ASM* *Xmem = src << SHIFT1* *Smem = src << SHIFT*	6-38
STLM *src*, *MMR*	5-27	*MMR = src* *mmr(MMR) = src*	6-38
STM *#lk*, *MMR*	5-27	*MMR = #lk* *mmr(MMR) = #lk*	6-38
ST *src*, *Ymem* || ADD *Xmem*, *dst*	5-27	*Ymem =* HI *(src)* [<< *ASM*] *|| dst = dst +Xmem*<<16	6-41
ST *src*, *Ymem* || LD *Xmem*, *dst* ST *src*, *Ymem* || LD *Xmem*, T	5-27	*Ymem =* HI*(src)* [<< *ASM*] *|| dst = Xmem* << 16 *Ymem =* HI *(src)* [<<*ASM*] *||* T *= Xmem*	6-40
ST *src*, *Ymem* || MAC *Xmem*, *dst* ST *src*, *Ymem* || MACR *Xmem*, *dst*	5-27	*Ymem =* HI *(src)* [<< *ASM*] *|| dst = dst +* T ** Xmem* *Ymem =* HI *(src)* [<< *ASM*] *|| dst +=* T ** Xmem* *Ymem =* HI *(src)* [<< *ASM*] *|| dst =* rnd *(dst +* T ** Xmem)*	6-41
ST *src*, *Ymem* || MAS *Xmem*, *dst* ST *src*, *Ymem* || MASR *Xmem*, *dst*	5-27	*Ymem =* HI *(src) [<< ASM]* *|| dst = dst –* T ** Xmem* *Ymem =* HI *(src)* [<< *ASM*] *|| dst –=* T ** Xmem* *Ymem =* HI *(src) [<< ASM]* *|| dst =* rnd *(dst –* T ** Xmem)*	6-41
ST *src*, *Ymem* || MPY *Xmem*, *dst*	5-27	*Ymem =* HI *(src) [<<ASM]* *|| dst =* T ** Xmem*	6-41
ST *src*, *Ymem* || SUB *Xmem*, *dst*	5-28	*Ymem =* HI *(src) [<< ASM]* *|| dst = Xmem* << 16 *– dst*	6-41

Mnemonic Instruction	Page	Equivalent Algebraic Instruction	Page
STRCD *Xmem*, *cond*	5-28	if *(cond) Xmem =* T	6-39
SUB *Smem, src* SUB *Smem, TS, src* SUB *Smem,* 16, *src*[,*dst*] SUB *Smem* [, *SHIFT*], *src*[,*dst*] SUB *Xmem, SHIFT1, src* SUB *Xmem, Ymem, dst* SUB *#lk* [,*SHIFT1*],*src*[,*dst*] SUB *#lk,* 16, *src*[,*dst*] SUB *src*[,*SHIFT*][,*dst*] SUB *src, ASM*[,*dst*]	5-28	*src = src – Smem* *src –= Smem* *src = src – Smem << TS* *src –= Smem << TS* *dst = src – Smem <<* 16 *dst –= Smem <<* 16 *dst = src – Smem* [<< SHIFT] *dst –= Smem* [<< SHIFT] *src = src – Xmem << SHIFT* *src –= Xmem << SHIFT* *dst = Xmem <<* 16 *– Ymem <<* 16 *dst = src – #lk [<< SHIFT]* *dst –= #lk [<< SHIFT]* *dst = src – #lk <<* 16 *dst –= #lk <<* 16 *dst = dst – src <<* SHIFT *dst –= src << SHIFT* *dst = dst – src << ASM* *dst –= src << ASM*	6-21
SUBB *Smem,src*	5-28	*src = src – Smem* – Borrow *src –= Smem* – Borrow	6-21
SUBC *Smem,src*	5-28	subc *(Smem, src)*	6-21
SUBS *Smem,src*	5-28	*src = src* – uns *(Smem)* *src –=* uns *(Smem)*	6-21
TRAP *K*	5-28	trap *(k)*	6-33
WRITA *Smem*	5-28	prog *(A) = Smem*	6-42
XC *n* , *cond* [,*cond* [,*cond*]]	5-29	if *(cond [,cond [,cond]])* execute *(n)* (n = 1 or 2)	6-36
XOR *Smem, src* XOR *#lk,* [*SHIFT1,*] *src* [,*dst*] XOR *src*, [*SHIFT*] [,*dst*]	5-29	*src = src ^ Smem* *src ^= Smem* *dst = src ^ #lk [<< SHIFT]* *dst ^= #lk[<< SHIFT]* *dst = src ^ #lk << 16* *dst ^= #lk << 16* *dst = dst ^ src[<< SHIFT]* *dst ^= src[<< SHIFT]*	6-29
XORM *#lk, Smem*	5-29	*Smem = Smem ^ #lk* *Smem ^= #lk*	6-29

5.3 Mnemonic Instruction Set Summary

Syntax	Description
ABDST *Xmem*, *Ymem*	**Absolute Distance** Compute the distance of two vectors, based on the absolute value.
ABS *src* [, *dst*]	**Absolute Value of Accumulator** Load the absolute value of the source accumulator into the destination accumulator. If no destination is given, load the absolute value into the source accumulator.
ADD *Smem*, *src* **ADD** *Smem*, *TS*, *src* **ADD** *Smem*, 16, *src* [, *dst*] **ADD** *Smem* [, *SHIFT*], *src* [, *dst*] **ADD** *Xmem*, *SHIFT1*, *src* **ADD** *Xmem*, *Ymem*, *dst* **ADD** *#lk* [, *SHIFT2*], *src* [, *dst*] **ADD** *src* [, *SHIFT*] [, *dst*] **ADD** *src*, *ASM* [, *dst*]	**Add to Accumulator** Add the value, shifted if indicated, to the value of the selected accumulator (A or B) or to the mode value (*Ymem*). Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
ADDC *Smem*, *src*	**Add to Accumulator With Carry** Add the value from the data memory location and the value of C to the source accumulator. Store the result in the accumulator.
ADDM *#lk*, *Smem*	**Add to Memory Long Immediate** Add the value from a data memory location to the memory value. Store the result in the data memory location specified by *Smem*.
ADDS *Smem*, *src*	**Add to Accumulator With Sign Extension Suppressed** Add the value from data memory to the source accumulator. Store the result in the accumulator.
AND *Smem*, *src* **AND** *#lk*, [*SHIFT1*,] *src* [, *dst*] **AND** *#lk*, *16*, *src* [, *dst*] **AND** *src* [, *SHIFT*] [, *dst*]	**AND With Accumulator** AND the data value or constant with the source accumulator (A or B). If a shift is specified, left shift the data value before the AND. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
ANDM *#lk*, *Smem*	**AND Memory With Long Immediate** AND the data memory value with a constant. Store the result in the data memory location specified by *Smem*.
B[D] *pmad*	**Branch Unconditionally** Pass control to the designated program memory address (*pmad*). For a delayed branch (BD), the one 2-word instruction or the two 1-word instructions following the branch instruction are fetched from program memory and executed before the branch is taken.

Syntax	Description
BACC[D] *src*	**Branch to Address Specified by Accumulator** Pass control to the address residing in the lower part of the source accumulator.
BANZ[D] *pmad, Sind*	**Branch on Auxiliary Register Not Zero** Pass control to the designated program memory address (*pmad*). If the value of the current auxiliary register designated by *Sind* is not equal to 0, branch to the specified program memory address; otherwise, add two to the program counter and store the result in the program counter.
BC[D] *pmad, cond* [, *cond* [, *cond*]]	**Branch Conditionally With Optional Delay** If the specified condition is met, pass control to the designated program memory address (*pmad*).
BIT *Xmem*, *bit_code*	**Test Bit** Copy the specified bit of the memory value into the TC bit of status register ST0.
BITF *Smem*, *#lk*	**Test Bit Field Specified by Immediate Value** Test the specified bit of the data memory value by masking the data value with an immediate value (*lk*).
BITT *Smem*	**Test Bit Specified by TREG** Copy the specified bit of the data memory value to the TC bit in status register ST0. The four LSBs of TREG contain a bit code that specifies which bit is copied.
CALA[D] *src*	**Call Subroutine at Location Specified by Accumulator** Increment the PC and push it onto the stack; then pass control to the address residing in the lower part of the source accumulator.
CALL[D] *pmad*	**Call Unconditionally With Optional Delay** Pass control to the specified program memory address (*pmad*). Push the return address onto the stack.
CC[D] *pmad cond* [, *cond* [, *cond*]]	**Call Conditionally With Optional Delay** If the specified conditions are met, push the return address onto the stack, then pass control to the program memory address (*pmad*).
CMPL *src* [, *dst*]	**Complement Accumulator** Complement the value of the source accumulator (logical inversion). The result is stored in the destination accumulator.
CMPM *Smem*, *lk*	**Compare Memory With Long Immediate** Compare the value to the constant specified in the instruction.

Syntax	Description
CMPR *CC*, AR*x*	**Compare Auxiliary Register With AR0** Compare the condition of the designated auxiliary register (AR*x*) to AR0 and place the result in the TC bit. The comparison is specified by the CC value.
CMPS *src*, *Smem*	**Compare Select Max and Store** Compare the values located in the 16 MSBs and the 16 LSBs of the source accumulator (considered 2s-complement values). Store the maximum value in the memory location. Shift TRN left one bit.
DADD *Lmem*, *src* [, *dst*]	**Double Precision/Dual Mode Add to Accumulator** Add the value of the source accumulator to the value. The value of C16 determines how the instruction is treated. The result is stored in the destination accumulator, if specified; otherwise, the result is stored in the source accumulator.
DADST *Lmem*, *dst*	**Double Precision Load With TREG Add/Dual 16-Bit Load With TREG Add/Subtract** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of TREG is left shifted 16 bits and ORed with itself. The resulting value is added to the long memory word. ❑ If C16 is set, the 16 MSBs of the long memory location are added to the value of TREG. At the same time, the value of TREG is subtracted from the 16 LSBs of the memory location. The result is stored in the destination accumulator.
DELAY *Smem*	**Memory Delay** Copy the value referenced by *Smem* into the next higher address.
DLD *Lmem*, *dst*	**Long Word Load to Accumulator** Load the destination accumulator with a 32-bit long operand value.
DRSUB *Lmem*, *src*	**Double Precision/Dual 16-Bit Subtract From Long Word** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of the source accumulator is subtracted from the long memory word. ❑ If C16 is set, the source accumulator high is subtracted from the 16 MSBs of the long memory location. At the same time, the source accumulator low is subtracted from the 16 LSBs of the memory location. The result is stored in the source accumulator.

Syntax	Description
DSADT *Lmem*, *dst*	**Long Load With TREG Add/Dual 16-Bit Load With TREG Subtract/Add** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of TREG is left shifted 16 bits and ORed with itself. The resulting value is subtracted from the 32-bit long memory word. ❑ If C16 is set, the value of TREG is subtracted from the 16 MSBs of the long memory location. At the same time, the value of TREG is added to the 16 LSBs of the memory location. The result is stored in the destination accumulator.
DST *src*, *Lmem*	**Store Accumulator in Long Word** Store the value of accumulator A or B in a long memory word.
DSUB *Lmem*, *src*	**Double Precision/Dual 16-Bit Subtract From Accumulator** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of the long memory word is subtracted from the source accumulator. ❑ If C16 is set, the value of the 16 MSBs of the long memory location is subtracted from the source accumulator high. At the same time, the 16 LSBs of the long memory location are subtracted from the source accumulator low. The result is stored in the accumulator.
DSUBT *Lmem*, *dst*	**Long Load With TREG Subtract/Dual 16-Bit Load With TREG Subtract** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of TREG is left-shifted 16 bits and ORed with itself. The resulting 32-bit value is subtracted from the 32-bit long memory word. ❑ If C16 is set, the value of TREG is subtracted from the 16 MSBs of the long memory word. At the same time, the value of TREG is subtracted from the 16 LSBs of the memory location. The result is stored in the destination accumulator.
EXP *src*	**Accumulator Exponent** Compute the exponent value defined as a signed 2s-complement value in the –8 to +31 range, and store the result in TREG.

Syntax	Description
FB[D] *extpmad*	**Far Branch Unconditionally** For a far call, pass control to the designated program memory address (*pmad*). For a delayed branch, the one 2-word instruction or the two 1-word instructions following the branch instruction are fetched from program memory and executed before the branch is taken.
FBACC[D] *src*	**Far Branch to Address Specified by Accumulator** For a far call, pass control to the address residing in the lower part of the source accumulator.
FCALA[D] *src*	**Far Call Subroutine at Location Specified by Accumulator** For a far call, increment the PC and push it onto the stack; then pass control to the address residing in the lower part of the source accumulator.
FCALL[D] *extpmad*	**Far Call Unconditionally With Optional Delay** For a far call, pass control to the specified program memory address (*pmad*). Push the return address onto the stack.
FIRS *Xmem*, *Ymem*, *pmad*	**Symmetrical Finite Impulse Response Filter** Multiply the accumulator A[32–16] by the program memory value (*pmad*). Accumulate the result in accumulator B. At the same time, add the two data memory values (*Xmem* and *Ymem*), left shift the result 16 bits, and store the result into accumulator A.
FRAME *K*	**Stack Pointer Immediate Offset** Perform a short immediate offset on the stack pointer.
FRET[D]	**Far Return With Optional Delay** For a far call, replace the program counter with the value addressed by the stack pointer. Increment the stack pointer by 1. Execution continues from this point.
FRETE[D]	**Enable Interrupts and Far Return From Interrupt With Optional Delay** For a far call, replace the program counter with the value addressed by the stack pointer. Execution continues from this point. RETE automatically clears the global interrupt mask bit (INTM in ST1).
IDLE *K*	**Idle Until Interrupt** Forces the executing program to halt until an unmasked interrupt or reset occurs. The PC is incremented only once, and the device remains in an idle state (power-down mode) until it is interrupted.
INTR *K*	**Software Interrupt** Transfer program control to the corresponding interrupt vector specified by K.

Syntax	Description
LD *Smem*, *dst* **LD** *Smem*, *TS*, *dst* **LD** *Smem*, 16, *dst* **LD** *Smem*, [*SHIFT* ,] *dst* **LD** *Xmem*, *SHIFT1*, *dst* **LD** *#K*, *dst* **LD** *#lk*, [*SHIFT1* ,] *dst* **LD** *#lk*, 16, *dst* **LD** *src*, *ASM* [, *dst*] **LD** *src*, [*SHIFT* ,] *dst*	**Load Accumulator With Shift** Load destination accumulator with a data memory value or an immediate value. This instruction has various shift capabilities. Accumulator-to-accumulator move with immediate shift or accumulator shift mode is supported.
	Load TREG or Status Register
LD *Smem*, T **LD** *Smem*, *DP* **LD** *Smem*, *ASM*	❑ Load a data memory value into TREG or the specified bit field of the status register (DP or ASM).
LD *#k9*, *DP* **LD** *#k5*, *ASM* **LD** *#k3*, *ARP*	❑ Load an immediate value into the specified bit field of the status register (DP, ASM, ARP).
LDM *MMR*, *dst*	**Load Memory-Mapped Register** Load the accumulator with a memory-mapped register value.
LD *Xmem*, *dst1* || **MAC** *Ymem* [, $\overline{dst}$] **LD** *Xmem*, *dst1* || **MACR** *Ymem* [, $\overline{dst}$]	**Multiply/Accumulate With/Without Rounding and Parallel Load** Multiply the value *Ymem* by the value of TREG and add the result of the multiplication to the accumulator that is not the destination accumulator ($\overline{dst}$). At the same time, load the destination accumulator high with value *Xmem*.
LD *Xmem*, *dst* || **MAS** *Ymem* [, $\overline{dst2}$] **LD** *Xmem*, *dst* || **MASR** *Ymem* [, $\overline{dst}$]	**Multiply/Subtract With/Without Rounding and Parallel Load** Multiply the value *Ymem* by the value of TREG and subtract the result of the multiplication from the accumulator that is not the destination accumulator ($\overline{dst}$). At the same time, load the destination accumulator high with the value *Xmem*.
LDR *Smem*, *dst*	**Load Memory Value in Accumulator High With Rounding** Load the data memory value into accumulator high.
LDU *Smem*, *dst*	**Load Memory Unsigned Value** Load the value into accumulator low. Guard bits and accumulator high are cleared.
LMS *Xmem*, *Ymem*	**Least Mean Square Instruction** Set up the basis for computing the least mean square. Replace the value of the accumulator B with the result of B plus *Xmem* multiplied by *Ymem*. Replace the value of accumulator A with the result of A plus *Xmem* left-shifted 16 bits and added to 2^{15}.

Syntax	Description
LTD *Smem*	**Load TREG and Memory Delay** Load the value into TREG, and copy the value to the next higher address.
	Multiply/Accumulate With/Without Rounding
MAC *Smem*, *src* **MACR** *Smem*, *src*	❑ Multiply TREG by the specified value and add the product to the source accumulator. Store the result in the source accumulator.
MAC *#lk*, *src* [, *dst*]	❑ Multiply TREG by the immediate value and add the product to the accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
MAC *Xmem*, *Ymem*, *src* [, *dst*] **MACR** *Xmem*, *Ymem*, *src* [, *dst*]	❑ Multiply *Xmem* and *Ymem* and add the product to the accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
MAC *Smem*, *#lk*, *src* [, *dst*]	❑ Multiply two values and add the product to the accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
	Multiply by Accumulator A and Accumulate
MACA *Smem*, [, B] **MACAR** *Smem*, [, B]	❑ Multiply the high bits of accumulator A by the specified value, and add the product to the value of accumulator B. Store the result in accumulator B.
MACA T, *src* [, *dst*] **MACAR** T, *src* [, *dst*]	❑ Multiply the high bits of accumulator A by the value of TREG, and add the product to the value of the source accumulator (A or B). Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
MACD *Smem*, *pmad*, *src*	**Multiply by Program Memory and Accumulate With Delay** Multiply two values. Add the product to the source accumulator value and store the result in that accumulator. The data memory value is copied into TREG and into the next higher address.
MACP *Smem*, *pmad*, *src*	**Multiply by Program Memory and Accumulate** Multiply two values. Add the product to the source accumulator value, and store the result in that accumulator. The data memory value is copied into TREG.
MACSU *Xmem*, *Ymem*, *src*	**Multiply Signed by Unsigned and Accumulate** Multiply an unsigned value (*Xmem*) by a signed value (*Ymem*), and add the product to the source accumulator. Store the result in the source accumulator. The unsigned value is stored in TREG.

Syntax	Description
MAR *Smem* **MAR** *ARn+0 **MAR** *ARn–0	**Modify Auxiliary Register** This instruction works in indirect addressing mode. The value of CMPT determines the execution of the instruction: ❑ If CMPT = 1 and AR*X* = AR0 or AR*X* = null, modify the auxiliary register pointed to by ARP. ARP is unchanged. If *X* is a nonnull value, then modify the auxiliary register and place the value X into ARP. ❑ If CMPT = 0, modify AR*X*, but do not change ARP. If direct addressing is used, MAR functions as a NOP.
	Multiply and Subtract
MAS *Smem*, *src* **MASR** *Smem*, *src*	❑ Multiply TREG by a value, and subtract the product from the accumulator. Store the result in the source accumulator.
MAS *Xmem*, *Ymem*, *src* [, *dst*] **MASR** *Xmem*, *Ymem*, *src* [, *dst*]	❑ Multiply two values (*Xmem* and *Ymem*), and subtract the product from the accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator. After the instruction, TREG contains the *Xmem* value.
	Multiply by Accumulator A and Subtract
MASA *Smem* [, B]	❑ Multiply the high bits of accumulator A by a value (*Smem*), and subtract the product from accumulator B. Store the result in accumulator B. After the instruction, TREG contains the *Smem* value.
MASA T, *src* [, *dst*] **MASAR** T, *src* [, *dst*]	❑ Multiply the high bits of accumulator A by the TREG value. Subtract the product from the source accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
MAX *dst*	**Accumulator Maximum** Compare the values of the accumulators, and store the maximum value in the destination accumulator.
MIN *dst*	**Accumulator Minimum** Compare the values of the accumulators, and store the minimum value in the destination accumulator.
	Multiply
MPY *Smem*, *dst* **MPYR** *Smem*, *dst* **MPY** *#lk*, *dst*	❑ Multiply the value of TREG by the value of the data memory location or by the constant. Store the result in the destination accumulator.
MPY *Smem*, *#lk*, *dst*	❑ Multiply the value of the data memory location by the constant *#lk*. Store the result in the destination accumulator.
MPY *Xmem*, *Ymem*, *dst*	❑ Multiply the value of the data memory location *Xmem* by the data memory location *Ymem*. Store the result in the destination accumulator.

Syntax	Description
	Multiply by Accumulator A
MPYA *dst*	❑ Multiply the high bits of accumulator A by the value of TREG. Store the result in the destination accumulator.
MPYA *Smem*	❑ Multiply the high bits of accumulator A by the value of the data memory location. Store the result in accumulator B. After this instruction, TREG contains the *Smem* value.
MPYU *Smem*, *dst*	**Multiply Unsigned** Multiply the value of TREG by the value of the data memory location *Smem*. Store the result in the destination accumulator.
MVDD *Xmem*, *Ymem*	**Move Data From Data Memory to Data Memory With X, Y Addressing** Move the value of the data memory location addressed by *Xmem* to the data memory location addressed by *Ymem*.
MVDK *Smem*, *dmad*	**Move Data From Data Memory to Data Memory With Destination Addressing** Move the value of the addressed data memory location (*Smem*) to a data memory location addressed by *dmad.*
MVDM *dmad*, *MMR*	**Move Data From Data Memory to Memory-Mapped Register** Move the value of the data memory location (*dmad*) to a memory-mapped register (*MMR*).
MVDP *Smem*, *pmad*	**Move Data From Data Memory to Program Memory** Move a value (*Smem*) to a program memory location addressed by value *dmad*.
MVKD *dmad*, *Smem*	**Move Data From Data Memory to Data Memory With Source Addressing** Move the value of the data memory location *dmad* to the value *Smem*.
MVMD *MMR*, *dmad*	**Move Data From Memory-Mapped Register to Data Memory** Move the value found in the memory-mapped register *MMR* to the value *dmad*.
MVMM *MMRx*, *MMRy*	**Move Data From Memory-Mapped Register to Memory-Mapped Register** Move the value of the memory-mapped register *MMRx* to the memory-mapped register *MMRy*.
MVPD *pmad*, *Smem*	**Move Data From Program Memory to Data Memory** Move the value found in the program memory address *pmad* to a data memory location addressed by *Smem*.

Syntax	Description
NEG *src* [, *dst*]	**Negate Accumulator** Negate (1s complement) the value of the source accumulator (A or B). The value is stored in the source accumulator or in the destination accumulator, if given.
NOP	**No Operation** No operation is performed. Only the program counter is incremented.
NORM *src* [, *dst*]	**Normalization** Allow single cycle normalization of the accumulator once the EXP instruction has been executed. The shift value is defined by TREG[5:0] and coded as a 2s-complement value.
	OR With Accumulator
OR *Smem*, *src*	❑ OR the 16 LSBs of the source accumulator with the value of the addressed data memory location. The result replaces the 16 LSBs of the source accumulator, leaving the rest of the accumulator unchanged.
OR *#lk* [, *SHIFT2*] , *src* [, *dst*] **OR** *#lk*, 16, *src* [, *dst*]	❑ OR the source accumulator with an immediate addressed value, left-shifted as indicated. Store in the destination accumulator, if specified; otherwise, store in the source accumulator.
OR *src* [, *SHIFT*] [, *dst*]	❑ OR the source accumulator with itself or the destination accumulator, if specified, left shifted as indicated. Store in the destination accumulator, if specified; otherwise, store in the source accumulator.
ORM *#lk*, *Smem*	**OR Memory With Constant** OR the value *Smem* with the constant *lk*. Store the result in *Smem*.
POLY *Smem*	**Polynomial Evaluation** Shift the value of the value *Smem* 16 bits to the left, and store the result in accumulator B. At the same time, multiply the value of accumulator A (17 MSBs) by the value of TREG, add it to accumulator B, and round the result. Store the result in accumulator A.
POPD *Smem*	**Pop Top of Stack to Data Memory** Move the value of the data memory addressed by the stack pointer to the memory specified by *Smem*. Increment the stack pointer by 1.
POPM *MMR*	**Pop Top of Stack to Memory-Mapped Register** Move the value of the data memory addressed by the stack pointer to the specified memory-mapped register. Increment the stack pointer by 1.
PORTR *PA*, *Smem*	**Read Data From Port** Read a value from an external I/O port (*PA*) into the specified data memory location (*Smem*).

Syntax	Description
PORTW *Smem*, *PA*	**Write Data to Port** Write a value to an external I/O port (*PA*) from the specified data memory location (*Smem*).
PSHD *Smem*	**Push Data Memory Value Onto Stack** Decrement the stack pointer by 1. Move the value of the data memory location (*Smem*) to the data memory location addressed by the stack pointer.
PSHM *MMR*	**Push Memory-Mapped Register Onto Stack** Decrement the stack pointer by 1. Move the value of the selected memory mapped register to the data memory location addressed by the stack pointer.
RC[D] *cond* [, *cond* [, *cond*]]	**Return Conditionally With Optional Delay** If the specified condition is met, return from subroutine. Replace the program counter with the data memory value addressed by the stack pointer and increment the stack pointer by 1. If the condition is not met, increment the PC by 1.
READA *Smem*	**Read Program Memory Addressed by Accumulator A** Transfer a 16-bit word from a program memory location to a data memory location (*Smem*). The program memory address is specified by the 16 LSBs of accumulator A.
RESET	**Software Reset** Put the DSP into a known state.
RET[D]	**Return With Optional Delay** Replace the program counter with the value addressed by the stack pointer. Increment the stack pointer by 1. Execution continues from this point.
RETE[D]	**Enable Interrupts and Return From Interrupt With Optional Delay** Replace the program counter with the value addressed by the stack pointer. Execution continues from this point. RETE automatically clears the global interrupt mask bit (INTM in ST1).
RETF[D]	**Enable Interrupts and Fast Return From Interrupt** Replace the program counter with the value addressed by the stack pointer. Execution continues at this point. RETF automatically sets the global interrupt mask bit (INTM in ST1).
RND ***src*** [, ***dst***]	**Round Accumulator** Rounds the content of *src* (either A or B) by adding 2^{15}. The rounded value is stored in *dst*.

Syntax	Description
ROL *src*	**Rotate Accumulator Left** Rotate the source accumulator left one bit.
ROLTC *src*	**Rotate Accumulator Left With TC** Rotate the source accumulator left one bit.
ROR *src*	**Rotate Accumulator Right** Rotate the source accumulator right one bit.
RPT *Smem* **RPT** *#K* **RPT** *#lk*	**Repeat Next Instruction** Set the repeat counter (RC) with the iteration value when RPT is executed. Load the value or constant into the repeat counter. The instruction following the RPT is repeated.
RPTB[D] *pmad*	**Block Repeat** The RPTB instruction allows a block of instructions to be repeated the number of times specified by the memory-mapped block repeat counter (BRC).
RPTZ *dst*, *#lk*	**Repeat Next Instruction With Clearing Accumulator** Clear the destination accumulator and load the value into the repeat counter register (RC). Repeat the instruction following the RPTZ instruction *n* times, where *n* is the value of the RC plus 1.
RSBX *N*, *SBIT*	**Reset Status Register Bit** Clear the specified bit (*SBIT*) of the status register.
SACCD *src*, *Xmem*, *cond*	**Store Accumulator Conditionally** If the condition is met, store the source accumulator (A or B) left-shifted as defined in *Xmem*. If the condition is not true, execute a read and a write at the *Xmem* address with the same value. Regardless of the condition, *Xmem* is always read and updated according to its modifier.
SAT *src*	**Saturate Accumulator** Regardless of the OVM value, saturate the source accumulator content on 32 bits.
SFTA *src*, *SHIFT* [, *dst*]	**Shift Accumulator Arithmetically** Shift the source accumulator arithmetically. Store the result in the destination accumulator, if specified; otherwise, replace the value of the source accumulator.
SFTC *src*	**Accumulator Conditional Shift** If the source accumulator has two significant sign bits, shift the accumulator left one bit. If there are two sign bits, TC is cleared; otherwise, TC is set to 1.

Syntax	Description
SFTL *src*, *SHIFT* [, *dst*]	**Shift Accumulator Logically** Shift the source accumulator logically. Store the result in the destination accumulator, if specified; otherwise, replace the value of the source accumulator.
SQDST *Xmem*, *Ymem*	**Square Distance** Used in repeat single mode, compute the distance between two vectors based on the square value.
SQUR *Smem*, *dst* **SQUR** A, *dst*	**Square** Square the value of *Smem*, or the 16 MSBs of accumulator A and store the result in the destination accumulator. Store the value of *Smem* in TREG.
SQURA *Smem*, *src*	**Square and Accumulate** Store the value of *Smem* in the T register. Square the value of *Smem* and add the result to the source accumulator. Store the result of the square/addition in the source accumulator.
SQURS *Smem*, *src*	**Square and Subtract** Store the value of *Smem* in TREG. Square the value of *Smem* and subtract the result from the source accumulator. Store the results of the square/subtraction in the source accumulator.
SRCCD *Xmem*, *cond*	**Store Block Repeat Counter Conditionally** If the condition is true, store the value of the BRC in *Xmem.* If the condition is false, execute a read and a write at the *Xmem* address with the same value. Regardless of the condition, *Xmem* is always updated and read according to its modifier.
SSBX *N*, *SBIT*	**Set Status Register Bit** Set the specified bit (*SBIT*) of the status register.
ST T, *Smem* **ST** *TRN*, *Smem* **ST** *#lk*, *Smem*	**Store T, TRN, Immediate Value Into Memory** Store the value of TREG, TRN, or the immediate value in data memory.
STH *src*, *Smem* **STH** *src*, *ASM*, *Smem* **STH** *src*, *SHIFT1*, *Xmem* **STH** *src*, [*SHIFT*,] *Smem*	**Store Accumulator High Into Memory** Store the 16 MSBs of the source accumulator in data memory (*Smem* or *Xmem*).
STL *src*, *Smem* **STL** *src*, *ASM*, *Smem* **STL** *src*, *SHIFT1*, *Xmem* **STL** *src*, [*SHIFT*,] *Smem*	**Store Accumulator Low Into Memory** Store the 16 LSBs of the source accumulator in data memory (*Smem* or *Xmem*).

Syntax	Description
STLM *src*, *MMR*	**Store Accumulator Low Into Memory-Mapped Register** Store the 16 LSBs of the source accumulator into the memory-mapped register. The upper nine bits of the effective address are cleared to 0, regardless of the current value of DP or the upper nine bits of AR*X*.
STM *#lk*, *MMR*	**Store Immediate Value Into Memory-Mapped Register** Store the value into the specified memory-mapped register or any memory location on data page 0 without modifying the DP field in status register ST0.
ST *src*, *Ymem* || **ADD** *Xmem*, *dst*	**Store Accumulator With Parallel Add** Store the value of the source accumulator shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, add the value of the accumulator that is not the destination accumulator to the 16-bit left-shifted data memory location *Xmem*, and store the result in the destination accumulator.
ST *src*, *Ymem* || **LD** *Xmem*, *dst* **ST** *src*, *Ymem* || **LD** *Xmem*, T	**Store Accumulator With Parallel Load** Store the value of the source accumulator shifted as defined by the ASM field of ST1 in the memory location *Ymem*. At the same time, load the value *Xmem* into the destination accumulator or TREG.
ST *src*, *Ymem* || **MAC** *Xmem*, *dst*	**Store Accumulator and Parallel Multiply/Accumulate With/Without Rounding**
ST *src*, *Ymem* || **MACR** *Xmem*, *dst*	Store the value of the source accumulator high shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, multiply the value *Xmem* by the value of TREG, add the result to the value of the destination accumulator, round if indicated, and store the final result in the destination accumulator.
ST *src*, *Ymem* || **MAS** *Xmem*, *dst*	**Store Accumulator and Parallel Multiply/Subtract With/Without Rounding**
ST *src*, *Ymem* || **MASR** *Xmem*, *dst*	Store the value of the source accumulator high shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, multiply the value *Xmem* by the value of TREG, subtract the result from the value of the destination accumulator, round if indicated, and store the result in the destination accumulator.
ST *src*, *Ymem* || **MPY** *Xmem*, *dst*	**Store Accumulator With Parallel Multiply** Store the value of the source accumulator high shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, multiply the value of TREG by the 16-bit dual addressed mode value *Xmem*, and store the result in the destination accumulator.

Syntax	Description
ST *src*, *Ymem* || **SUB** *Xmem*, *dst*	**Store Accumulator With Parallel Subtract** Store the source accumulator high shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, left shift the value *Xmem* 16 bits, subtract the value of the accumulator that is not the destination accumulator, and store the result in the destination accumulator.
STRCD *Xmem*, *cond*	**Store TREG Conditionally** If the condition is true, store the value of TREG in *Xmem*. Regardless of the condition, *Xmem* is always read and updated according to its modifier. If the condition is true, execute a read and a write at the *Xmem* address with the same value.
SUB *Smem*, *src* **SUB** *Smem*, *TS*, *src* **SUB** *Smem*, 16, *src*, [, *dst*] **SUB** *Smem*, [*SHIFT* ,] *src* [, *dst*] **SUB** *Xmem*, *SHIFT1*, *src* **SUB** *Xmem*, *Ymem*, *dst* **SUB** *#lk*, [*SHIFT1* ,] *src* [, *dst*] **SUB** *#lk*, 16, *src* [, *dst*] **SUB** *src* [, *SHIFT*] [, *dst*] **SUB** *src*, *ASM* [, *dst*]	**Subtract From Accumulator** Subtract the value of a data memory location, a constant, or the source accumulator (A or B), shifted if indicated, from the value of the source accumulator (A or B) or from the value *Ymem*. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
SUBB *Smem*, *src*	**Subtract From Accumulator With Borrow** Subtract the value of the mode value from the value of the source accumulator. Subtract the logical inversion of C from the result. The final result is stored in the source accumulator.
SUBC *Smem*, *src*	**Subtract From Accumulator Conditionally** Subtract the mode value, left shifted 15 bits, from the value of the source accumulator. If the result is greater than 0, left shift the result by 1, add 1, and store the final result in the source accumulator. Otherwise, left shift the value of the source accumulator by 1 and store the result in the source accumulator.
SUBS *Smem*, *src*	**Subtract From Accumulator With Sign-Extension Suppressed** Subtract the mode value from the value of the source accumulator. The result is stored in the source accumulator.
TRAP *K*	**Software Interrupt** Transfer program control to the interrupt routine specified by *K*. Decrement the stack pointer by 1. Push the value of PC + 1 onto the data memory addressed by the stack pointer.
WRITA *Smem*	**Write Data Memory Addressed by Accumulator A** Transfer a 16-bit word from a data memory location (*Smem*) to a program memory location specified by the 16 LSBs of accumulator A.

Syntax	Description
XC *n, cond* [, *cond* [, *cond*]]	**Execute Conditionally** The value of *n* and the selected conditions determine the execution of the instruction: ❑ If n = 1 and the condition is met, fetch and execute the next 1-word instruction. ❑ If n = 2 and the condition is met, fetch and execute the next two 1-word instructions or the next one 2-word instruction. ❑ If the condition is not met, execute one or two NOP instructions.
XOR *Smem*, *src* **XOR** *#lk*, [*SHIFT1* ,] *src* [, *dst*] **XOR** *src*, [*SHIFT*] [, *dst*]	**Exclusive-OR With Accumulator** Exclusive-OR the mode value, shifted if indicated, with the value of the source accumulator (A or B). Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
XORM *#lk*, *Smem*	**Exclusive-OR Memory With Constant** Exclusive-OR the value of the data memory location with the value *lk*. Store the result in the data memory location.

Chapter 6

Algebraic Instruction Set Summary

The TMS320C54x device supports general-purpose instructions as well as arithmetic-intensive instructions that are particularly suited for DSP and other numeric-intensive applications. There are two instruction sets—algebraic and mnemonic. These two sets perform the same functions but with very different syntax.

This chapter contains a summary of the algebraic instruction set. Table entries show the syntax for the instruction and describes the instruction operation. Section 6.1, *Using the Summary Tables*, shows a sample table entry and describes the abbreviations used in the table. Section 6.2, *Algebraic and Mnemonic Instruction Sets Cross-Reference,* on page 6-5 cross references the mnemonic instruction set to the algebraic instruction set.

This chapter does not cover topics such as opcodes, instruction timing, or addressing modes.

Topic **Page**

6.1 Using the Summary Tables

To help you read the summary table, this section provides examples of table entries and lists of acronyms.

6.1.1 Table Entry Example

This is how the AND function algebraic instruction appears in the summary table:

Example 6–1. Table Entry for an Algebraic Instruction

Syntax	Description
src = src & *Smem* *src* &= *Smem* *dst = src* & #*lk* [<< *SHIFT1*] *dst* &= #*lk* [<< *SHIFT1*] *dst = src* & #*lk* << 16 *dst* &= #*lk* << 16 *dst = dst* & *src* [<< *SHIFT*] *dst* &= *src* [<< *SHIFT*]	**AND With Accumulator** AND the data value or constant with the source accumulator (A or B). If a shift is specified, left shift the data value before the AND. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.

6.1.2 Table Entry Explained

The syntax column lists the syntax for the AND function instruction. Alternative syntax is shown in the lines following the first syntax. Abbreviations used in syntax are found in subsection 6.1.3, *Symbols and Acronyms*, on page 6-2.

The description column briefly describes how the instruction functions. Often, an instruction functions slightly differently with different parameters. For complete information about an instruction, see the *TMS320C54x User's Guide*.

6.1.3 Symbols and Acronyms

The following table lists the instruction set symbols and acronyms used throughout this chapter.

Table 6–1. Symbols and Acronyms Used in the Instruction Set Summary

Symbol	Definition
( )	Contents of
[ ]	Optional items; the brackets are not part of the syntax
#	Prefix of constants used in immediate addressing

Table 6–1. Symbols and Acronyms Used in the Instruction Set Summary (Continued)

Symbol	Definition
| |	Parallel instructions
ARP	Auxiliary register pointer
ARX	Auxiliary register (AR0–AR7)
ASM	Accumulator shift mode (from ST1)
Borrow	A literal that specifies to subtract with borrow
BRC	Block repeat counter
Carry	Carry bit
C16	Dual 16-bit/double-precision bit
CC	Condition code
CMPT	Compatibility mode bit
cond	Conditional expression
data	Data memory access which can be incremented under a single repeat instruction
dbl	32-bit access
dmad	16-bit data immediate addressed value
DP	Data page pointer
dst	Destination accumulator (A or B)
$\overline{\text{dst}}$	Accumulator opposite the previous dst
dst1, dst2	If *dst1* = A, then *dst2* = B If *dst1* = B, then *dst2* = A
dual	Dual access
extpmad	23-bit immediate program-memory address
HI	High half of the register
INTM	Interrupt mask bit
K	Short immediate value (less than 9 bits)
k3	3-bit immediate value ($0 \leq k3 \leq 7$)
k5	5-bit immediate value ($-16 \leq k5 \leq 15$)
k9	9-bit immediate value ($0 \leq k9 \leq 511$)
lk	Long immediate value (16 bits)
Lmem	32-bit single-addressed mode value (direct or indirect) (long word addressing)
mmr, MMR	Memory-mapped register

Table 6–1. Symbols and Acronyms Used in the Instruction Set Summary (Continued)

Symbol	Definition
MMRx, MMRy	Memory-mapped register, AR0–AR7 or SP
n	Operand for the XC instruction indicating the number of instructions to conditionally execute ❑ 1 = One instruction executes ❑ 0 = Two instructions execute
N	Status register for RSBX/SSBX instruction; either 0 or 1
OVM	Overflow mode bit
PA	16-bit port immediate addressed value
PC	Program counter
pmad	16-bit program immediate addressed value
port	I/O port access
prog	Program memory access
RC	Repeat counter
rnd	Round the expression
SBIT	Status register bit for RSBX/SSBX instruction; $0 \leq$ SBIT ≤ 15
SHIFT	Shift value in –16 to +15 range
SHIFT1	Shift value in 0 to 15 range
SHIFT2	Shift value in 0 to 16 range
Sind	Single indirect address mode
Smem	16-bit single-addressed mode value (direct or indirect)
SP	Stack pointer register
src	Source accumulator (A or B)
T	A literal that specifies to use TREG as operand
TC	Test/control flag bit
TREG	Temporary register
TRN	Transition register
TS	Shift value held in TREG (–16, +31 range)
unx	Unsigned operand
Xmem	16-bit dual addressed mode value (indirect only) used in dual operand instructions and some single operand instructions
Ymem	16-bit dual addressed mode value (indirect only) used mainly in dual operands instructions

6.2 Algebraic and Mnemonic Instruction Sets Cross-Reference

The following tables cross reference the algebraic instruction set with the mnemonic instruction set. A summary of an instruction is shown on the indicated page. See the *TMS320C54x User's Guide* for detailed information about the instruction sets.

To translate mnemonic code to algebraic code use the translator utility (see Chapter 13, *Mnemonic-to-Algebraic Translator Description*).

Table 6–2. Add Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
src = src + Smem *src += Smem* *src = src + Smem << TS* *src += Smem << TS* *dst = src + Smem* << 16 *dst += Smem <<* 16 *dst = src + Smem* [<< *SHIFT*] *dst += Smem* [<< *SHIFT*] *src = src + Xmem << SHIFT1* *src += Xmem << SHIFT1* *dst = Xmem* << *16 + Ymem <<* 16 *dst = src + #lk* [<< *SHIFT1*] *dst += #lk* [<< *SHIFT1*] *dst = src + #lk* << 16 *dst += #lk* << 16 *dst = dst + src* [<< *SHIFT*] *dst += src* [<< *SHIFT*] *dst = dst + src << ASM* *dst += src << ASM*	6-20	ADD *Smem, src* ADD *Smem, TS, src* ADD *Smem,* 16, *src* [, *dst*] ADD *Smem* [, *SHIFT*], *src* [, *dst*] ADD *Xmem, SHIFT1*, *src* ADD *Xmem, Ymem, dst* ADD *#lk* [, *SHIFT2*], *src* [, *dst*] ADD *#lk*, 16, *src* [, *dst*] ADD *src* [, *SHIFT*] [, *dst*] ADD *src, ASM* [, *dst*]	5-14
src = src + Smem + Carry *src += Smem + Carry*	6-20	ADDC *Smem, src*	5-14
Smem = Smem + #lk *Smem += #lk*	6-20	ADDM *#lk*, *Smem*	5-14
src = src + uns *(Smem)* *src* += uns *(Smem)*	6-20	ADDS *Smem, src*	5-14

Table 6–3. Subtract Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
src = src – Smem *src –= Smem* *src = src – Smem << TS* *src –= Smem << TS* *dst = src – Smem <<* 16 *dst –= Smem <<* 16 *dst = src – Smem* [*<< SHIFT*] *dst –= Smem* [*<< SHIFT*] *src = src – Xmem << SHIFT* *src –= Xmem << SHIFT* *dst = Xmem <<* 16 *– Ymem <<* 16 *dst = src – #lk [<< SHIFT]* *dst –= #lk [<< SHIFT]* *dst = src – #lk <<* 16 *dst –= #lk <<* 16 *dst = dst – src* [*<< SHIFT*] *dst –= src* [*<< SHIFT*] *dst = dst – src << ASM* *dst –= src << ASM*	6-21	SUB *Smem, src* SUB *Smem, TS, src* SUB *Smem,* 16, *src*[*, dst*] SUB *Smem* [*, SHIFT*] *, src*[*, dst*] SUB *Xmem, SHIFT1, src* SUB *Xmem, Ymem, dst* SUB *#lk* [*, SHIFT1*] *,src*[*, dst*] SUB *#lk,* 16, *src*[*, dst*] SUB *src*[*, SHIFT*][*, dst*] SUB *src, ASM*[*, dst*]	5-28
src = src – Smem – Borrow *src –= Smem* – Borrow	6-21	SUBB *Smem,src*	5-28
subc *(Smem, src)*	6-21	SUBC *Smem,src*	5-28
src = src – uns *(Smem)* *src –=* uns *(Smem)*	6-21	SUBS *Smem,src*	5-28

Table 6–4. Multiply Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
dst = T ** Smem* *dst* = T ** #lk* *dst* = rnd *(*T ** Smem)* *dst = Smem * #lk [,*T = *Smem]* *dst = Xmem * Ymem [,*T = *Xmem]*	6-22	MPY *Smem*, *dst* MPY *Xmem*, *Ymem*, *dst* MPY *Smem*, #*lk*, *dst* MPY #*lk*, *dst* MPYR *Smem*, *dst*	5-21
dst = T * HI *(A)* *B = Smem* * HI *(A)* [*,* T = *Smem*]	6-22	MPYA *dst* MPYA *Smem*	5-22
dst = T * uns *(Smem)*	6-22	MPYU *Smem*, *dst*	5-22
*dst = Smem * Smem* [*,* T = *Smem*] *dst* = square*(Smem)* [*,* T = *Smem*] *dst* = HI *(A)* * HI *(A)* *dst* = square *(*HI *(A))*	6-22	SQUR *Smem, dst* SQUR *A, dst*	5-26

Table 6–5. Multiply-Accumulate or Multiply-Subtract Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
dst = (*src* + T * *Smem*) *src* += T * *Smem* *dst* = rnd *(src* + T * *Smem)* *dst* = *src* + T * *#lk* *dst* += T * *#lk* *dst* = (*src* + *Xmem* * *Ymem*) [, T = *Xmem*] *dst* += *Xmem* * *Ymem* [, T = *Xmem*] *dst* = rnd *(src* + *Xmem* * *Ymem)* [, T = *Xmem*] *dst* = *src* + *Smem* * *#lk* [, T = *Smem*] *dst* += *Smem* * *#lk* [, T = *Smem*]	6-23	MAC *Smem*, *src* MAC *Xmem*, *Ymem*, *src* [, *dst*] MAC *#lk*, *src* [, *dst*] MAC *Smem*, *#lk*, *src* [, *dst*] MACR *Smem*, *src* MACR *Xmem*, *Ymem*, *src* [, *dst*]	5-20
B = *B* + *Smem* * HI *(A)* [, T = *Smem*] *B* += *Smem* * HI *(A)* [, T = *Smem*] *B* = rnd *(B* + *Smem* * HI *(A))* [, T = *Smem*] *dst* = *src* + T * HI *(A)* *dst* += T * HI *(A)* *dst* = rnd *(src* + T * HI *(A))*	6-23	MACA *Smem* [, *B*] MACA T, *src* [, *dst*] MACAR *Smem* [, *B*] MACAR T, *src* [, *dst*]	5-20
macd *(Smem, pmad, src)*	6-23	MACD *Smem*, *pmad*, *src*	5-20
macp *(Smem, pmad, src)*	6-23	MACP *Smem*, *pmad*, *src*	5-20
src = *src* + uns *(Xmem)* * *Ymem* [, T = *Xmem*] *src* += uns *(Xmem)* * *Ymem* [, T = *Xmem*]	6-23	MACSU *Xmem*, *Ymem*, *src*	5-20
src = *src* – T * *Smem* *src* –= T * *Smem* *src* = rnd *(src* – T * *Smem)* *dst* = *src* – *Xmem* * *Ymem* [, T = *Xmem*] *dst* –= *Xmem* * *Ymem* [, T = *Xmem*] *dst* = rnd *(src* – *Xmem* * *Ymem)* [, T=*Xmem*]	6-23	MAS *Smem, src* MAS *Xmem*, *Ymem*, *src* [, *dst*] MASR *Smem, src* MASR *Xmem*, *Ymem*, *src* [, *dst*]	5-21
B = *B* – *Smem* * HI *(A)* [, T = *Smem*] *B* –= *Smem* * HI *(A)* [, T = *Smem*] *dst* = *src* – T * HI *(A)* *dst* –= T * HI *(A)* *dst* = rnd *(src* – T * HI *(A))*	6-23	MASA *Smem* [, *B*] MASA T, *src* [, *dst*] MASAR T, *src* [, *dst*]	5-21
src = *src* + square *(Smem)* [, T=*Smem*] *src* += square *(Smem)* [, T = *Smem*] *src* = *src* + *Smem* * *Smem* [, T=*Smem*] *src* += *Smem* * *Smem* [, T = *Smem*]	6-23	SQURA *Smem, src*	5-26
src = *src* – square *(Smem)* [, T=*Smem*] *src* –= square *(Smem)* [, T = *Smem*] *src* = *src* – *Smem* * *Smem* [, T=*Smem*] *src* –= *Smem* * *Smem* [, T = *Smem*]	6-23	SQURS *Smem, src*	5-26

Table 6–6. Double (32-bit Operand) Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
dst = src + dbl *(Lmem)* *dst* += dbl *(Lmem)* *dst = src* + dual *(Lmem)* *dst* += dual *(Lmem)*	6-25	DADD *Lmem,* src [*, dst*]	5-16
dst = dadst *(Lmem,* T*)*	6-25	DADST *Lmem*, *dst*	5-16
dst = dbl *(Lmem)* *dst* = dual *(Lmem)*	6-25	DLD *Lmem*, *dst*	5-16
src = dbl *(Lmem) – src* *src* = dual *(Lmem) – src*	6-25	DRSUB *Lmem*, *src*	5-16
dst = dsadt *(Lmem,* T*)*	6-25	DSADT *Lmem*, *dst*	5-17
dbl *(Lmem) = src* dual *(Lmem) = src*	6-25	DST *src*, *Lmem*	5-17
src = src – dbl *(mem)* *src* –= dbl *(Lmem)* *src = src* – dual *(mem)* *src* –= dual *(Lmem)*	6-25	DSUB *Lmem*, *src*	5-17
dst = dbl *(Lmem)* – T *dst* = dual *(Lmem)* – T	6-25	DSUBT *Lmem*, *dst*	5-17

Table 6–7. Application-Specific Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
abdst (*Xmem, Ymem*)	6-27	ABDST *Xmem*, *Ymem*	5-14
dst = | *src*|	6-27	ABS *src* [*, dst*]	5-14
dst = ~*src*	6-27	CMPL *src* [, *dst*]	5-15
delay *(Smem*)	6-27	DELAY *Smem*	5-16
T = exp *(src*)	6-27	EXP *src*	5-17
firs *(Xmem, Ymem, pmad)*	6-27	FIRS *Xmem*, *Ymem*, *pmad*	5-18
lms *(Xmem, Ymem)*	6-27	LMS *Xmem*, *Ymem*	5-19
dst = max (A, B)	6-27	MAX *dst*	5-21
dst = min (A, B)	6-27	MIN *dst*	5-21
dst = –*src*	6-27	NEG *src* [, *dst*]	5-23
dst = src << TS *dst* = norm *(src,TS)*	6-27	NORM *src* [, *dst*]	5-23
poly *(Smem)*	6-27	POLY *Smem*	5-23
dst = rnd (*src*)	6-27	RND *src* [, *dst*]	5-24
saturate *(src)*	6-27	SAT *src*	5-25
sqdst *(Xmem, Ymem)*	6-27	SQDST *Xmem*, *Ymem*	5-26

Table 6–8. AND Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
src = src & *Smems* *src* &= *Smem* *dst = src* & *#lk* [<< *SHIFT1*] *dst* &= *#lk* [<< *SHIFT1*] *dst = src* & *#lk* << 16 *dst* &= *#lk* << 16 *dst = dst* & *src* [<< *SHIFT*] *dst* &= *src* [<< *SHIFT*]	6-28	AND *Smem*, *src* AND *#lk* [, *SHFT]*, *src* [, *dst*] AND *#lk*, 16, *src* [, *dst*] AND *src* [, *SHIFT*] [, *dst*]	5-14
*Smem = Sme*m & *#lk* *Smem* & = *#lk*	6-28	ANDM *#lk*, *Smem*	5-14

Table 6–9. OR Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
src = src | *Smem* *src* |= *Smem* *dst = src* | *#lk* [*<< SHIFT*] *dst* |= *#lk* [*<< SHIFT]* *dst = src* | *#lk* << 16 *dst* |= *#lk* << 16 *dst = dst* | *src* [*<< SHIFT*] *dst* |= *src* [*<< SHIFT*]	6-29	OR *Smem, src* OR *#lk* [, *SHIFT2*], *src* [, *dst*] OR *#lk*, 16, *src* [, *dst*] OR *src* [*, SHIFT*] [, *dst*]	5-23
Smem = Smem | *#lk* *Smem* |= *#lk*	6-29	ORM *#lk, Smem*	5-23

Table 6–10. XOR Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
src = src ^ Smem *src ^= Smem* *dst = src ^ #lk [<< SHIFT]* *dst ^= #lk[<< SHIFT]* *dst = src ^ #lk* << 16 *dst ^= #lk* << 16 *dst = dst ^ src[<< SHIFT]* *dst ^= src[<< SHIFT]*	6-29	XOR *Smem, src* XOR *#lk,* [*SHIFT1*,] *src* [, *dst*] XOR *#lk,* 16, *src* [, *dst*] XOR *src*, [*SHIFT*] [, *dst*]	5-29
Smem = Smem ^ #lk *Smem ^= #lk*	6-29	XORM *#lk, Smem*	5-29

Table 6–11. Shift Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
src = src || *Carry*	6-30	ROL *src*	5-25
roltc *(src)*	6-30	ROLTC *src*	5-25
src = src // Carry	6-30	ROR *src*	5-25
dst = src << C SHIFT	6-30	SFTA *src*, *SHIFT* [, *dst*]	5-25
shiftc *(src)*	6-30	SFTC *src*	5-25
dst = src << SHIFT	6-30	SFTL *src*, *SHIFT* [, *dst*]	5-26

Table 6–12. Test Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
TC = bit *(Xmem, bit_code)*	6-31	BIT *Xmem*, *bit_code*	5-15
TC = bitf (*Smem*, *#lk)*	6-31	BITF *Smem*, *#lk*	5-15
TC = bitt (*Smem)*	6-31	BITT *Smem*	5-15
TC = (*Smem* == *#lk*)	6-31	CMPM *Smem, lk*	5-15
TC = (AR0 == AR*x*) (==, <, >, !=) *TC* = (AR0 > AR*x*) *TC* = (AR0 < AR*x*) *TC* = (AR0 != AR*x*)	6-31	CMPR *CC*, AR*x*	5-16

Table 6–13. Branch Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
[d]goto *pmad*	6-32	B[D] *pmad*	5-14
[d]goto *src*	6-32	BACC[D] *src*	5-15
if (*Sind* != *0)* [d]goto *pmad*	6-32	BANZ[D] *pmad*, *Sind*	5-15
if (cond [, *cond*] [, *cond*]) [d]goto *pmad*	6-32	BC[D] *pmad*, *cond* [, *cond* [, *cond*]]	5-15
[d]fgoto *pmad*	6-32	FB[D] *extpmad*	5-18
[d]fgoto *src*	6-32	FBACC[D] *src*	5-18

Table 6–14. Call Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
[d]call *src*	6-33	CALA[D] *src*	5-15
[d]call *pmad*	6-33	CALL[D] *pmad*	5-15
if (*cond* [, *cond*] [, *cond*] *)* [d]call *pmad*	6-33	CC[D] *pmad*, *cond* [, *cond* [, *cond*]]	5-15
[d]fcall *src*	6-33	FCALA[D] *src*	5-18
[d]fcall *pmad*	6-33	FCALL[D] *extpmad*	5-18

Table 6–15. Interrupt Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
int *(K)* (0 <= K <= 31)	6-33	INTR *K*	5-18
trap *(k)*	6-33	TRAP *K*	5-28

Table 6–16. Return Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
[d]freturn	6-34	FRET[D]	5-18
[d]freturn_enable	6-34	FRETE[D]	5-18
if *(cond* [*, cond*] [*, cond*] *)* [d]return	6-34	RC[D] *cond* [, *cond* [, *cond*]]	5-24
[d]return	6-34	RET[D]	5-24
[d]return_enable	6-34	RETE[D]	5-24
[d]return_fast	6-34	RETF[D]	5-24

Table 6–17. Repeat Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
repeat *(Smem)* repeat *(#k)* (RC=#k) (0<=k<=255) repeat *(#lk)* (RC=#lk) (0<=lk<=65535)	6-35	RPT *Smem* RPT #*K* RPT #*lk*	5-25
[d]blockrepeat *(pmad)*	6-35	RPTB *pmad* RPTBD *pmad*	5-25
repeat *(#lk), dst = 0*	6-35	RPTZ *dst, #lk*	5-25

Table 6–18. Stack Manipulating Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
SP = SP + K (–128 <= K <= 127) *SP += K* (–128 <= K <= 127)	6-35	FRAME *K*	5-18
Smem = pop()	6-35	POPD *Smem*	5-23
MMR = pop() *mmr*(*MMR*) *=* pop ( )	6-35	POPM *MMR*	5-23
push *(Smem)*	6-35	PSHD *Smem*	5-24
push *(MMR)* push *(mmr(MMR))*	6-35	PSHM *MMR*	5-24

Table 6–19. Miscellaneous Program Control Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
idle *(K)* (0 <= K <= 3)	6-36	IDLE *K*	5-18
mar *(Smem)* AR*n* = AR*n* + AR0 AR*n* += AR0 AR*n* = AR*n* – AR0 AR*n* –= AR0	6-36	MAR *Smem* MAR *AR*n*+0 MAR *AR*n*–0	5-21
nop	6-36	NOP	5-23
reset	6-36	RESET	5-24
SBIT = 0 *ST(N,SBIT)=0*	6-36	RSBX *N*, *SBIT*	5-25
SBIT = 1 ST *(N,SBIT)=1*	6-36	SSBX *N*, *SBIT*	5-26
if *(cond [,cond [,cond]])* execute *(n)* (n = 1 or 2)	6-36	XC *n* , *cond* [, *cond* [, *cond*]]	5-29

Table 6–20. Load Instructions

Algebraic Instruction		Page	Equivalent Mnemonic Instruction	Page
dst = Smem		6-37	LD *Smem*, *dst*	5-19
dst = Smem << TS			LD *Smem*, *TS*, *dst*	
dst = Smem << 16			LD *Smem, 16, dst*	
dst = Smem [*<< SHIFT*]			LD *Smem* [,*SHIFT*] , *dst*	
dst = Xmem [*<< SHIFT1*]			LD *Xmem*, *SHIFT1*, *dst*	
dst = #K	(0 <= K <= 255)		LD *#K, dst*	
dst = #lk [*<< SHIFT1*]			LD *#lk* [,*SHIFT1*] , *dst*	
dst = #lk << 16			LD *#lk*, 16, *dst*	
dst = src << ASM			LD *src*, *ASM* [, *dst*]	
dst = src [*<< SHIFT*]			LD *src* [,*SHIFT*] [, *dst*]	
T = *Smem*		6-37	LD *Smem,* T	5-19
DP = Smem			LD *Smem, DP*	
ASM = Smem	(ASM = Smem*[*0:4*]*)		LD *#k9*, *DP*	
DP = #k9	(0 <= k9 <= 511)		LD *#k5*, *ASM*	
ASM = #k5	(–16 <= k5 <= 15)		LD *#k3*, *ARP*	
ARP = #k3	(0 <= k3 <= 7)		LD *Smem, ASM*	
dst = MMR		6-37	LDM *MMR, dst*	5-19
dst = mmr(MMR)				
dst = rnd *(Smem)*		6-37	LDR *Smem*, *dst*	5-19
dst = uns *(Smem)*		6-37	LDU *Smem*, *dst*	5-19
ltd *(Smem)*		6-37	LTD *Smem*	5-20

Table 6–21. Store Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
Smem = T *Smem = TRN* *Smem = #lk*	6-38	ST T, *Smem* ST *TRN*, *Smem* ST *#lk*, *Smem*	5-26
Smem = HI *(src)* *Smem = (src) << ASM* *Xmem* = HI *(src) << SHIFT1* *Smem* = HI *(src) << SHIFT*	6-38	STH *src*, *Smem* STH *src*, *ASM*, *Smem* STH *src*, *SHIFT1*, *Xmem* STH *src* [,*SHIFT*], *Smem*	5-26
Smem = src *Smem = src << ASM* *Xmem = src << SHIFT1* *Smem = src << SHIFT*	6-38	STL *src*, *Smem* STL *src*, *ASM*, *Smem* STL *src*, *SHIFT1*, *Xmem* STL *src* [,*SHIFT*], *Smem*	5-26
MMR = src *mmr(MMR) = src*	6-38	STLM *src*, *MMR*	5-27
MMR = #lk *mmr(MMR) = #lk*	6-38	STM *#lk*, *MMR*	5-27

Table 6–22. Conditional Store Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
cmps(src,Smem)	6-39	CMPS *src, Smem*	5-16
if *(cond) Xmem* = HI *(src) << ASM*	6-39	SACCD *src, Xmem, cond*	5-25
if *(cond) Xmem = BRC*	6-39	SRCCD *Xmem*, *cond*	5-26
if *(cond) Xmem* = T	6-39	STRCD *Xmem*, *cond*	5-28

Table 6–23. Parallel Load and Store Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
dst = *Xmem* [<< *16*] || *dst* = *dst* + T**Ymem* *dst* = *Xmem* [<< *16*] || *dst* += T**Ymem* *dst* = *Xmem* [<< *16*] || dst = rnd *(dst* + T * *Ymem)*	6-40	LD *Xmem*, *dst* || MAC *Ymem* [, *dst*] LD *Xmem*, *dst* || MACR *Ymem* [, *dst*]	5-19
dst = *Xmem* [<< *16*] *|| dst* = *dst* – T * *Ymem* *dst* = *Xmem* [<< *16*] *|| dst –=* T * *Ymem* *dst* = *Xmem* [<< *16*] *|| dst* = rnd (*dst* – T * *Ymem*)	6-40	LD *Xmem*, *dst* || MAS *Ymem* [, *dst*] LD *Xmem*, *dst* || MASR *Ymem*[, *dst*]	5-19
Ymem = HI *(src)* [<< *ASM*] *|| dst = Xmem* << 16 *Ymem* = HI *(src)* [<<*ASM*] *||* T *= Xmem*	6-40	ST *src*, *Ymem* || LD *Xmem*, *dst* ST *src*, *Ymem* || LD *Xmem*, T	5-27

Table 6–24. Parallel Store and Multiply Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
Ymem = HI *(src)* [<< *ASM*] *|| dst = dst +* T * *Xmem* *Ymem* = HI *(src)* [<< *ASM*] *|| dst +=* T * *Xmem* *Ymem* = HI *(src)* [<< *ASM*] *|| dst* = rnd *(dst +* T * *Xmem)*	6-41	ST *src*, *Ymem* || MAC *Xmem*, *dst* ST *src*, *Ymem* || MACR *Xmem*, *dst*	5-27
Ymem = HI *(src) [<< ASM]* *|| dst = dst* – T * *Xmem* *Ymem* = HI *(src)* [<< *ASM*] *|| dst –=* T * *Xmem* *Ymem* = HI *(src) [<< ASM]* *|| dst* = rnd *(dst* – T * *Xmem)*	6-41	ST *src*, *Ymem* || MAS *Xmem*, *dst* ST *src*, *Ymem* || MASR *Xmem*, *dst*	5-27
Ymem = HI *(src) [<<ASM]* *|| dst =* T * *Xmem*	6-41	ST *src*, *Ymem* || MPY *Xmem*, *dst*	5-27

Table 6–25. Parallel Store and Add/Subtract Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
Ymem = HI *(src)* [<< *ASM*] || *dst* = *dst* +*Xmem*<<16	6-41	ST *src*, *Ymem* || ADD *Xmem*, *dst*	5-27
Ymem = HI *(src) [<< ASM]* || *dst* = *Xmem* << 16 – *dst*	6-41	ST *src*, *Ymem* || SUB *Xmem*, *dst*	5-28

Table 6–26. Miscellaneous Load-Type and Store-Type Instructions

Algebraic Instruction	Page	Equivalent Mnemonic Instruction	Page
Ymem = *Xmem*	6-42	MVDD *Xmem*, *Ymem*	5-22
data *(dmad)* = *Smem*	6-42	MVDK *Smem*, *dmad*	5-22
MMR = data *(dmad)* *mmr(MMR)* = data *(dmad)*	6-42	MVDM *dmad*, *MMR*	5-22
prog *(pmad)* = *Smem*	6-42	MVDP *Smem*, *pmad*	5-22
Smem = data *(dmad)*	6-42	MVKD *dmad*, *Smem*	5-22
data *(dmad)* = *MMR* data *(dmad)* = *mmr(MMR)*	6-42	MVMD *MMR*, *dmad*	5-22
MMRy = *MMRx* (0 <= x,y <= 8) *mmr(MMRy)* = *mmr(MMRx)*	6-42	MVMM *MMRx*, *MMRy*	5-22
Smem = prog *(pmad)*	6-42	MVPD *pmad*, *Smem*	5-22
Smem = port *(PA)*	6-42	PORTR *PA*, *Smem*	5-23
port *(PA)* = *Smem*	6-42	PORTW *Smem, PA*	5-24
Smem = prog *(A)*	6-42	READA *Smem*	5-24
prog *(A)* = *Smem*	6-42	WRITA *Smem*	5-28

6.3 Algebraic Instruction Set Summary

The ’54x algebraic instruction set can be divided into four basic types of operations:

- Arithmetic operations
- Logical operations
- Program-control operations
- Load and store operations

In this summary, each of the types of operations is divided into smaller groups of instructions with similar functions.

6.3.1 Arithmetic Operations

The arithmetic operation instructions are grouped as follows:

Instruction Group	See	On Page
Add	Table 6–27	6-20
Subtract	Table 6–28	6-21
Multiply	Table 6–29	6-22
Multiply accumulate	Table 6–30	6-23
Multiply subtract	Table 6–30	6-23
Double	Table 6–31	6-25
Application specific	Table 6–32	6-27

6.3.2 Logical Operations

The logical operation instructions are grouped as follows:

Instruction Group	See	On Page
AND	Table 6–33	6-28
OR	Table 6–34	6-29
XOR	Table 6–35	6-29
Shift	Table 6–36	6-30
Test	Table 6–37	6-31

6.3.3 Program Control Operations

The program control operation instructions are grouped as follows:

Instruction Group	See	On Page
Branch	Table 6–38	6-32
Call	Table 6–39	6-33
Interrupt	Table 6–40	6-33
Return	Table 6–41	6-34
Repeat	Table 6–42	6-35
Stack manipulation	Table 6–43	6-35
Miscellaneous	Table 6–44	6-36

6.3.4 Load and Store Operations

The load and store operation instructions are grouped as follows:

Instruction Group	See	On Page
Load	Table 6–45	6-37
Store	Table 6–46	6-38
Conditional store	Table 6–47	6-39
Parallel load and store	Table 6–48	6-40
Parallel store and multiply	Table 6–49	6-41
Parallel store and add/subtract	Table 6–50	6-41
Miscellaneous	Table 6–51	6-42

6.3.5 Algebraic Instruction Set Summary Tables

Table 6–27. Add Instructions

Syntax	Description
src = src + Smem *src += Smem* *src = src + Smem << TS* *src += Smem << TS* *dst = src + Smem* << 16 *dst += Smem <<* 16 *dst = src + Smem* [<< *SHIFT*] *dst += Smem* [<< *SHIFT*] *src = src + Xmem << SHIFT1* *src += Xmem << SHIFT1* *dst = Xmem* << 16 + *Ymem <<* 16 *dst = src + #lk* [<< *SHIFT1*] *dst += #lk* [<< *SHIFT1*] *dst = src + #lk* << 16 *dst += #lk* << 16 *dst = dst + src* [<< *SHIFT*] *dst += src* [<< *SHIFT*] *dst = dst + src << ASM* *dst += src << ASM*	**Add to Accumulator** Add the value, shifted if indicated, to the value of the selected accumulator (A or B) or to the mode value (*Ymem*). Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
src = src + Smem + Carry *src += Smem + Carry*	**Add to Accumulator With Carry** Add the value from the data memory location and the value of C to the source accumulator. Store the result in the accumulator.
Smem = Smem + #lk *Smem += #lk*	**Add to Memory Long Immediate** Add the value from a data memory location to the memory value. Store the result in the data memory location specified by *Smem*.
src = src + uns *(Smem)* *src* += uns *(Smem)*	**Add to Accumulator With Sign Extension Suppressed** Add the value from data memory to the source accumulator. Store the result in the accumulator.

Table 6–28. Subtract Instructions

Syntax	Description
src = src – Smem *src –= Smem* *src = src – Smem << TS* *src –= Smem << TS* *dst = src – Smem* << 16 *dst –= Smem* << 16 *dst* = src – Smem [<< *SHIFT*] *dst* –= Smem [<< *SHIFT*] *src = src – Xmem << SHIFT* *src –= Xmem << SHIFT* *dst = Xmem* << 16 – *Ymem* << 16 *dst = src – #lk* [<< *SHIFT*] *dst –= #lk* [<< *SHIFT*] *dst = src – #lk* << 16 *dst –= #lk* << 16 *dst = dst – src* [<< *SHIFT*] *dst –= src* [<< *SHIFT*] *dst = dst – src << ASM* *dst –= src << ASM*	**Subtract From Accumulator** Subtract the value of a data memory location, a constant, or the source accumulator (A or B), shifted if indicated, from the value of the source accumulator (A or B) or from the value *Ymem*. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
src = src – Smem – Borrow *src –= Smem* – Borrow	**Subtract From Accumulator With Borrow** Subtract the value of the mode value from the value of the source accumulator. Subtract the logical inversion of C from the result. The final result is stored in the source accumulator.
subc *(Smem, src)*	**Subtract From Accumulator Conditionally** Subtract the mode value, left shifted 15 bits, from the value of the source accumulator. If the result is greater than 0, left shift the result by 1, add 1, and store the final result in the source accumulator. Otherwise, left shift the value of the source accumulator by 1 and store the result in the source accumulator.
src = src – uns *(Smem)* *src –=* uns *(Smem)*	**Subtract From Accumulator With Sign-Extension Suppressed** Subtract the mode value from the value of the source accumulator. The result is stored in the source accumulator.

Table 6–29. Multiply Instructions

Syntax	Description
	Multiply
dst = T * *Smem* *dst* = rnd (T * *Smem*) *dst* = T * *#lk*	❑ Multiply the value of TREG by the value of the data memory location or by the constant. Store the result in the destination accumulator.
dst = *Smem* * *#lk* [, T = *Smem*]	❑ Multiply the value of the data memory location by the constant *#lk*. Store the result in the destination accumulator.
dst = *Xmem* * *Ymem* [, T = *Xmem*]	❑ Multiply the value of the data memory location *Xmem* by the data memory location *Ymem*. Store the result in the destination accumulator.
	Multiply by Accumulator A
dst = T * HI (*A*)	❑ Multiply the high bits of accumulator A by the value of TREG. Store the result in the destination accumulator.
B = *Smem* * HI (*A*) [, T = *Smem*]	❑ Multiply the high bits of accumulator A by the value of the data memory location. Store the result in accumulator B. After this instruction, TREG contains the *Smem* value.
dst = T * uns (*Smem*)	**Multiply Unsigned** Multiply the value of TREG by the value of the data memory location *Smem*. Store the result in the destination accumulator.
dst = *Smem* * *Smem* [, T = *Smem*] *dst* = square (*Smem*) [, T = *Smem*] *dst* = HI (*A*) * HI (*A*) *dst* = square (HI (*A*))	**Square** Square the value of *Smem*, or the 16 MSBs of accumulator A and store the result in the destination accumulator. Store the value of *Smem* in TREG.

Table 6–30. Multiply-Accumulate or Multiply-Subtract Instructions

Syntax	Description
	Multiply/Accumulate With/Without Rounding
dst = src + T ** Smem* *dst +=* T ** Smem* *dst =* rnd *(src* + T * *Smem)*	❑ Multiply TREG by the specified value and add the product to the source accumulator. Store the result in the source accumulator.
dst = src + T ** #lk* *dst* += T ** #lk*	❑ Multiply TREG by the immediate value and add the product to the accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
*dst = src + Xmem * Ymem* [, T = *Xmem*] *dst += Xmem * Ymem* [, T = *Xmem*] *dst* = rnd *(src + Xmem * Ymem)* [, T = *Xmem*]	❑ Multiply *Xmem* and *Ymem* and add the product to the accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
*dst = src + Smem * #lk* [, T = *Smem*] *dst += Smem * #lk* [, T = *Smem*]	❑ Multiply two values and add the product to the accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
	Multiply by Accumulator A and Accumulate
*B = B + Smem ** HI *(A)* [, T = *Smem*] *B +=* *Smem ** HI *(A)* [, T = *Smem*] *B* = rnd *(B + Smem ** HI *(A))* [, T = *Smem*]	❑ Multiply the high bits of accumulator A by the specified value, and add the product to the value of accumulator B. Store the result in accumulator B.
dst = src + T * HI *(A)* *dst* += T * HI *(A)* *dst* = rnd *(src* + T * HI *(A))*	❑ Multiply the high bits of accumulator A by the value of TREG, and add the product to the value of the source accumulator (A or B). Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
macd *(Smem, pmad, src)*	**Multiply by Program Memory and Accumulate With Delay** Multiply two values. Add the product to the source accumulator value and store the result in that accumulator. The data memory value is copied into TREG and into the next higher address.
macp *(Smem, pmad, src)*	**Multiply by Program Memory and Accumulate** Multiply two values. Add the product to the source accumulator value, and store the result in that accumulator. The data memory value is copied into TREG.

Table 6–30. Multiply-Accumulate or Multiply-Subtract Instructions (Continued)

Syntax	Description
src = src + uns *(Xmem) * Ymem* [,T = *Xmem*] *src* += uns *(Xmem) * Ymem* [,T = *Xmem*]	**Multiply Signed by Unsigned and Accumulate** Multiply an unsigned value (*Xmem*) by a signed value (*Ymem*), and add the product to the source accumulator. Store the result in the source accumulator. The unsigned value is stored in TREG.
	Multiply and Subtract
src = src – T ** Smem* *src* –= T ** Smem* *src =* rnd *(src* – T ** Smem)*	❑ Multiply TREG by a value, and subtract the product from the accumulator. Store the result in the source accumulator.
*dst = src – Xmem * Ymem* [, T = *Xmem*] *dst* –= *Xmem * Ymem* [, T = *Xmem*] *dst =* rnd *(src – Xmem * Ymem)* [, T=*Xmem*]	❑ Multiply two values (*Xmem* and *Ymem*), and subtract the product from the accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator. After the instruction, TREG contains the *Xmem* value.
	Multiply by Accumulator A and Subtract
*B = B – Smem ** HI *(A)* [, T = *Smem*] *B* –= *Smem ** HI *(A)* [, T = *Smem*]	❑ Multiply the high bits of accumulator A by a value (*Smem*), and subtract the product from accumulator B. Store the result in accumulator B. After the instruction, TREG contains the *Smem* value.
dst = src – T * HI *(A)* *dst* –= T * HI *(A)* *dst =* rnd *(src* – T * HI *(A))*	❑ Multiply the high bits of accumulator A by the TREG value. Subtract the product from the source accumulator. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
src = src + square *(Smem)* [, T=*Smem*] *src* += square *(Smem)* [, T = *Smem*] *src = src + Smem * Smem* [, T=*Smem*] *src* += *Smem * Smem* [, T = *Smem*]	**Square and Accumulate** Store the value of *Smem* in the T register. Square the value of *Smem* and add the result to the source accumulator. Store the result of the square/addition in the source accumulator.
src = src – square *(Smem)* [, T=*Smem*] *src* –= square *(Smem)* [, T = *Smem*] *src = src – Smem * Smem* [, T=*Smem*] *src* –= *Smem * Smem* [, T = *Smem*]	**Square and Subtract** Store the value of *Smem* in TREG. Square the value of *Smem* and subtract the result from the source accumulator. Store the results of the square/subtraction in the source accumulator.

Table 6–31. Double (32-bit Operand) Instructions

Syntax	Description
dst = src + dbl *(Lmem)* *dst* += dbl *(Lmem)* *dst = src* + dual *(Lmem)* *dst* += dual *(Lmem)*	**Double Precision/Dual Mode Add to Accumulator** Add the value of the source accumulator to the value. The value of C16 determines how the instruction is treated. The result is stored in the destination accumulator, if specified; otherwise, the result is stored in the source accumulator.
dst = dadst *(Lmem,* T *)*	**Double Precision Load With TREG Add/Dual 16-Bit Load With TREG Add/Subtract** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of TREG is left-shifted 16 bits and ORed with itself. The resulting value is added to the long memory word. ❑ If C16 is set, the 16 MSBs of the long memory location are added to the value of TREG. At the same time, the value of TREG is subtracted from the 16 LSBs of the memory location. The result is stored in the destination accumulator.
dst = dbl *(Lmem)* *dst* = dual *(Lmem)*	**Long Word Load to Accumulator** Load the destination accumulator with a 32-bit long operand value.
src = dbl *(Lmem) – src* *src* = dual *(Lmem) – src*	**Double Precision/Dual 16-Bit Subtract From Long Word** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of the source accumulator is subtracted from the long memory word. ❑ If C16 is set, the source accumulator high is subtracted from the 16 MSBs of the long memory location. At the same time, the source accumulator low is subtracted from the 16 LSBs of the memory location. The result is stored in the source accumulator.

Table 6–31. Double (32-bit Operand) Instructions (Continued)

Syntax	Description
dst = dsadt *(Lmem,* T*)*	**Long Load With TREG Add/Dual 16-Bit Load With TREG Subtract/Add** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of TREG is left-shifted 16 bits and ORed with itself. The resulting value is subtracted from the 32-bit long memory word. ❑ If C16 is set, the value of TREG is subtracted from the 16 MSBs of the long memory location. At the same time, the value of TREG is added to the 16 LSBs of the memory location. The result is stored in the destination accumulator.
dbl *(Lmem) = src* dual *(Lmem) = src*	**Store Accumulator in Long Word** Store the value of accumulator A or B in a long memory word.
src = src – dbl(mem) *src* –= dbl *(Lmem)* *src = src – dual(mem)* *src* –= dual *(Lmem)*	**Double Precision/Dual 16-Bit Subtract From Accumulator** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of the long memory word is subtracted from the source accumulator. ❑ If C16 is set, the value of the 16 MSBs of the long memory location is subtracted from the source accumulator high. At the same time, the 16 LSBs of the long memory location are subtracted from the source accumulator low. The result is stored in the accumulator.
dst = dbl *(Lmem)* – T *dst* = dual *(Lmem)* – T	**Long Load With TREG Subtract/Dual 16-Bit Load With TREG Subtract** The value of C16 determines the execution of the instruction: ❑ If C16 is not set, the value of TREG is left-shifted 16 bits and ORed with itself. The resulting 32-bit value is subtracted from the 32-bit long memory word. ❑ If C16 is set, the value of TREG is subtracted from the 16 MSBs of the long memory word. At the same time, the value of TREG is subtracted from the 16 LSBs of the memory location. The result is stored in the destination accumulator.

Table 6–32. Application-Specific Instructions

Syntax	Description
abdst(*Xmem, Ymem*)	**Absolute Distance** Compute the distance of two vectors, based on the absolute value.
dst = |*src*|	**Absolute Value of Accumulator** Load the absolute value of the source accumulator into the destination accumulator. If no destination is given, load the absolute value into the source accumulator.
dst = ~*src*	**Complement Accumulator** Complement the value of the source accumulator (logical inversion). The result is stored in the destination accumulator.
delay *(Smem*)	**Memory Delay** Copy the value referenced by *Smem* into the next higher address.
T = exp *(src*)	**Accumulator Exponent** Compute the exponent value defined as a signed 2s-complement value in the –8 to +31 range, and store the result in TREG.
firs *(Xmem, Ymem, pmad)*	**Symmetrical Finite Impulse Response Filter** Multiply the accumulator A[32–16] by the program memory value (*pmad*). Accumulate the result in accumulator B. At the same time, add the two data memory values (*Xmem* and *Ymem*), left shift the result 16 bits, and store the result into accumulator A.
lms *(Xmem, Ymem)*	**Least Mean Square Instruction** Set up the basis for computing the least mean square. Replace the value of accumulator B with the result of B plus *Xmem* multiplied by *Ymem*. Replace the value of accumulator A with the result of A plus *Xmem* left-shifted 16 bits and added to 2^{15}.
dst = max(A, B)	**Accumulator Maximum** Compare the values of the accumulators, and store the maximum value in the destination accumulator.
dst = min(A, B)	**Accumulator Minimum** Compare the values of the accumulators, and store the minimum value in the destination accumulator.
dst = –*src*	**Negate Accumulator** Negate (1s complement) the value of the source accumulator (A or B). The value is stored in the source accumulator or in the destination accumulator, if given.

Table 6–32. Application-Specific Instructions (Continued)

Syntax	Description
dst = src << TS *dst =* norm *(src, TS)*	**Normalization** Allow single cycle normalization of the accumulator once the EXP instruction has been executed. The shift value is defined by TREG[5:0] and coded as a 2s-complement value.
poly *(Smem)*	**Polynomial Evaluation** Shift the value of the value *Smem* 16 bits to the left, and store the result in accumulator B. At the same time, multiply the value of accumulator A (17 MSBs) by the value of TREG, add it to accumulator B, and round the result. Store the result in accumulator A.
dst = rnd (*src*)	**Round Accumulator** Rounds the content of *src* (either A or B) by adding 2^{15}. The rounded value is stored in *dst*.
saturate *(src)*	**Saturate Accumulator** Regardless of the OVM value, saturate the source accumulator content on 32 bits.
sqdst *(Xmem, Ymem)*	**Square Distance** Used in repeat single mode, compute the distance between two vectors based on the square value.

Table 6–33. AND Instructions

Syntax	Description
src = src & *Smem* *src* &= *Smem* *dst = src* & *#lk* [<< *SHIFT1*] *dst* &= *#lk* [<< *SHIFT1*] *dst = src* & *#lk* << 16 *dst* &= *#lk* << 16 *dst = dst* & *src* [<< *SHIFT*] *dst* &= *src* [<< *SHIFT*]	**AND With Accumulator** AND the data value or constant with the source accumulator (A or B). If a shift is specified, left shift the data value before the AND. Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
*Smem = Sme*m & *#lk* *Smem* &= *#lk*	**AND Memory With Long Immediate** AND the data memory value with a constant. Store the result in the data memory location specified by *Smem*.

Table 6–34. OR Instructions

Syntax	Description
	OR With Accumulator
src = src | *Smem* *src* |= *Smem*	❑ OR the 16 LSBs of the source accumulator with the value of the addressed data memory location. The result replaces the 16 LSBs of the source accumulator, leaving the rest of the accumulator unchanged.
dst = src | *#lk* [*<< SHIFT*] *dst* |= *#lk* [*<< SHIFT]* *dst = src* | #*lk* << 16 *dst* |= #*lk* << 16	❑ OR the source accumulator with an immediate addressed value, left shifted as indicated. Store in the destination accumulator, if specified; otherwise, store in the source accumulator.
dst = dst | *src* [*<< SHIFT*] *dst* |= *src* [*<< SHIFT*]	❑ OR the source accumulator with itself or the destination accumulator, if specified, left shifted as indicated. Store in the destination accumulator, if specified; otherwise, store in the source accumulator.
Smem = Smem | *#lk* *Smem* |= *#lk*	**OR Memory With Constant** OR the value *Smem* with the constant *lk*. Store the result in *Smem*.

Table 6–35. XOR Instructions

Syntax	Description
src = src ^ Smem *src ^= Smem* *dst = src ^ #lk [<< SHIFT]* *dst ^= #lk [<< SHIFT]* *dst = src ^#lk* << 16 *dst ^= #lk* << 16 *dst = dst ^ src [<< SHIFT]* *dst ^= src [<< SHIFT]*	**Exclusive-OR With Accumulator** Exclusive-OR the mode value, shifted if indicated, with the value of the source accumulator (A or B). Store the result in the destination accumulator, if specified; otherwise, store the result in the source accumulator.
Smem = Smem ^ #lk *Smem ^= #lk*	**Exclusive-OR Memory With Constant** Exclusive-OR the value of the data memory location with the value *lk*. Store the result in the data memory location.

Table 6–36. Shift Instructions

Syntax	Description
src = src \\ Carry	**Rotate Accumulator Left** Rotate the source accumulator left one bit.
roltc *(src)*	**Rotate Accumulator Left With TC** Rotate the source accumulator left one bit.
src = src // Carry	**Rotate Accumulator Right** Rotate the source accumulator right one bit.
dst = src << C SHIFT	**Shift Accumulator Arithmetically** Shift the source accumulator arithmetically. Store the result in the destination accumulator, if specified; otherwise, replace the value of the source accumulator.
shiftc *(src)*	**Accumulator Conditional Shift** If the source accumulator has two significant sign bits, shift the accumulator left one bit. If there are two sign bits, TC is cleared; otherwise, TC is set to 1.
dst = src << SHIFT	**Shift Accumulator Logically** Shift the source accumulator logically. Store the result in the destination accumulator, if specified; otherwise, replace the value of the source accumulator.

Table 6–37. Test Instructions

Syntax		Description
TC = bit *(Xmem, bit_code)*		**Test Bit** Copy the specified bit of the memory value into the TC bit of status register ST0.
TC = bitf (*Smem*, *#lk)*		**Test Bit Field Specified by Immediate Value** Test the specified bit of the data memory value by masking the data value with an immediate value (*lk*).
TC = bitt (*Smem)*		**Test Bit Specified by TREG** Copy the specified bit of the data memory value to the TC bit in status register ST0. The four LSBs of TREG contain a bit code that specifies which bit is copied.
TC = (*Smem == #lk*)		**Compare Memory With Long Immediate** Compare the value to the constant specified in the instruction.
TC = (AR0 == AR*x*) *TC* = (AR0 > AR*x*) *TC* = (AR0 < AR*x*) *TC* = (AR0 != AR*x*)	(==, <, >, !=)	**Compare Auxiliary Register With AR0** Compare the condition of the designated auxiliary register (AR*x*) to AR0 and place the result in the TC bit. The comparison is specified by the CC value.

6.3.6 Program Control Operations

Table 6–38. Branch Instructions

Syntax	Description
[d]goto *pmad*	**Branch Unconditionally** Pass control to the designated program memory address (*pmad*). For a delayed branch ([d]goto), the one 2-word instruction or the two 1-word instructions following the branch instruction are fetched from program memory and executed before the branch is taken.
[d]goto *src*	**Branch to Address Specified by Accumulator** Pass control to the address residing in the lower part of the source accumulator.
if (*Sind* != *0*) [d]goto *pmad*	**Branch on Auxiliary Register Not Zero** Pass control to the designated program memory address (*pmad*). If the value of the current auxiliary register designated by *Sind* is not equal to 0, branch to the specified program memory address; otherwise, add two to the program counter and store the result in the program counter.
if (*cond* [, *cond* [*cond*]]) [d]goto *pmad*	**Branch Conditionally With Optional Delay** If the specified condition is met, pass control to the designated program memory address (*pmad*).
far [d]goto *extpmad*	**Far Branch Unconditionally** For a far call, pass control to the designated program memory address (*pmad*). For a delayed branch, the one 2-word instruction or the two 1-word instructions following the branch instruction are fetched from program memory and executed before the branch is taken.
far [d]goto *src*	**Far Branch to Address Specified by Accumulator** For a far call, pass control to the address residing in the lower part of the source accumulator.

Table 6–39. Call Instructions

Syntax	Description
[d]call *src*	**Call Subroutine at Location Specified by Accumulator** Increment the PC and push it onto the stack; then pass control to the address residing in the lower part of the source accumulator.
[d]call *extpmad*	**Call Unconditionally With Optional Delay** Pass control to the specified program memory address (*pmad*). Push the return address onto the stack.
if (*cond* [, *cond* [, *cond*]]) [d]call *pmad*	**Call Conditionally With Optional Delay** If the specified conditions are met, push the return address onto the stack, then pass control to the program memory address (*pmad*).
far [d]call *src*	**Far Call Subroutine at Location Specified by Accumulator** For a far call, increment the PC and push it onto the stack; then pass control to the address residing in the lower part of the source accumulator.
far [d]call *extpmad*	**Far Call Unconditionally With Optional Delay** For a far call, pass control to the specified program memory address (*pmad*). Push the return address onto the stack.

Table 6–40. Interrupt Instructions

Syntax	Description
int *(K)* (0 <= K <= 31)	**Software Interrupt** Transfer program control to the corresponding interrupt vector specified by K.
trap *(k)*	**Software Interrupt** Transfer program control to the interrupt routine specified by *K*. Decrement the stack pointer by 1. Push the value of PC + 1 onto the data memory addressed by the stack pointer.

Table 6–41. Return Instructions

Syntax	
far [d]return	**Far Return With Optional Delay** For a far call, replace the program counter with the value addressed by the stack pointer. Increment the stack pointer by 1. Execution continues from this point.
far [d]return_enable	**Enable Interrupts and Far Return From Interrupt With Optional Delay** For a far call, replace the program counter with the value addressed by the stack pointer. Execution continues from this point. RETE automatically clears the global interrupt mask bit (INTM in ST1).
if (*cond* [, *cond* [, *cond*]]) [d]return	**Return Conditionally With Optional Delay** If the specified condition is met, return from subroutine. Replace the program counter with the data memory value addressed by the stack pointer and increment the stack pointer by 1. If the condition is not met, increment the PC by 1.
[d]return	**Return With Optional Delay** Replace the program counter with the value addressed by the stack pointer. Increment the stack pointer by 1. Execution continues from this point.
[d]return_enable	**Enable Interrupts and Return From Interrupt With Optional Delay** Replace the program counter with the value addressed by the stack pointer. Execution continues from this point. RETE automatically clears the global interrupt mask bit (INTM in ST1).
[d]return_fast	**Enable Interrupts and Fast Return From Interrupt** Replace the program counter with the value addressed by the stack pointer. Execution continues at this point. RETF automatically sets the global interrupt mask bit (INTM in ST1).

Table 6–42. Repeat Instructions

Syntax	Description
repeat *(Smem)* repeat *(#K)* (RC=#k) (0<=k<=255) repeat *(#lk)* (RC=#lk) (0<=lk<=65535)	**Repeat Next Instruction** Set the repeat counter (RC) with the iteration value when RPT is executed. Load the value or constant into the repeat counter. The instruction following the RPT is repeated.
[d]blockrepeat *(pmad)*	**Block Repeat** The RPTB instruction allows a block of instructions to be repeated the number of times specified by the memory-mapped block repeat counter (BRC).
repeat *(#lk), dst = 0*	**Repeat Next Instruction With Clearing Accumulator** Clear the destination accumulator and load the value into the repeat counter register (RC). Repeat the instruction following the RPTZ instruction *n* times, where *n* is the value of the RC plus 1.

Table 6–43. Stack-Manipulating Instructions

Syntax	
SP = SP + K (–128 <= K <= 127) *SP += K* (–128 <= K <= 127)	**Stack Pointer Immediate Offset** Perform a short immediate offset on the stack pointer.
Smem = pop( )	**Pop Top of Stack to Data Memory** Move the value of the data memory addressed by the stack pointer to the memory specified by *Smem*. Increment the stack pointer by 1.
MMR = pop( ) *mmr*(*MMR*) = pop( )	**Pop Top of Stack to Memory-Mapped Register** Move the value of the data memory addressed by the stack pointer to the specified memory-mapped register. Increment the stack pointer by 1.
push *(Smem)*	**Push Data Memory Value Onto Stack** Decrement the stack pointer by 1. Move the value of the data memory location (*Smem*) to the data memory location addressed by the stack pointer.
push *(MMR)* push *(mmr(MMR))*	**Push Memory-Mapped Register Onto Stack** Decrement the stack pointer by 1. Move the value of the selected memory mapped register to the data memory location addressed by the stack pointer.

Table 6–44. Miscellaneous Program Control Instructions

Syntax	Description
idle *(K)* (0 <= K <= 3)	**Idle Until Interrupt** Forces the executing program to halt until an unmasked interrupt or reset occurs. The PC is incremented only once, and the device remains in an idle state (power-down mode) until it is interrupted.
mar *(Smem)* AR*n* = AR*n* + AR0 AR*n* += AR0 AR*n* = AR*n* – AR0 AR*n* –= AR0	**Modify Auxiliary Register** This instruction works in indirect addressing mode. The value of CMPT determines the execution of the instruction: ❑ If CMPT = 1 and AR*X* = AR0 or AR*X* = null, modify the auxiliary register pointed to by ARP. ARP is unchanged. If *X* is a nonnull value, then modify the auxiliary register and place the value X into ARP. ❑ If CMPT = 0, modify AR*X*, but do not change ARP. If direct addressing is used, MAR functions as a NOP.
nop	**No Operation** No operation is performed. Only the program counter is incremented.
reset	**Software Reset** Put the DSP into a known state.
SBIT = 0 ST*(N,SBIT)=0*	**Reset Status Register Bit** Clear the specified bit (*SBIT*) of the status register.
SBIT = 1 ST*(N,SBIT)=1*	**Set Status Register Bit** Set the specified bit (*SBIT*) of the status register.
if *(cond [,cond [,cond]])* execute *(n)* (n = 1 or 2)	**Execute Conditionally** The value of *n* and the selected conditions determine the execution of the instruction: ❑ If n = 1 and the condition is met, fetch and execute the next 1-word instruction. ❑ If n = 2 and the condition is met, fetch and execute the next two 1-word instructions or the next one 2-word instruction. ❑ If the condition is not met, execute one or two NOP instructions.

Table 6–45. Load Instructions

Syntax	Description
dst = Smem *dst = Smem << TS* *dst = Smem <<* 16 *dst = Smem* [*<< SHIFT*] *dst = Xmem* [*<< SHIFT1*] *dst = #K* (0 <= K <= 255) *dst = #lk* [*<< SHIFT1*] *dst = #lk <<* 16 *dst = src << ASM* *dst = src* [*<< SHIFT*]	**Load Accumulator With Shift** Load destination accumulator with a data memory value or an immediate value. This instruction has various shift capabilities. Accumulator–to-accumulator move with immediate shift or accumulator shift mode is supported.
T = *Smem* *DP = Smem* *ASM = Smem* (ASM = Smem[0:4]) *DP = #k9* (0 <= Kk9<= 511) *ASM = #k5* (–16 <= Kk5<= 15) *ARP = #k3* (0 <= k3 <= 7)	**Load TREG or Status Register** ❑ Load a data memory value into TREG or the specified bit field of the status register (DP or ASM). ❑ Load an immediate value into the specified bit field of the status register (DP, ASM, ARP).
dst = MMR	**Load Memory Mapped Register** Load the accumulator with a memory-mapped register value.
dst = rnd *(Smem)*	**Load Memory Value in Accumulator High With Rounding** Load the data memory value into accumulator high.
dst = uns *(Smem)*	**Load Memory Unsigned Value** Load the value into accumulator low. Guard bits and accumulator high are cleared.
ltd *(Smem)*	**Load TREG and Memory Delay** Load the value into TREG, and copy the value to the next higher address.

Table 6–46. Store Instructions

Syntax	Description
Smem = T *Smem = TRN* *Smem = #lk*	**Store T, TRN, Immediate Value Into Memory** Store the value of TREG, TRN, or the immediate value in data memory.
Smem = HI *(src)* *Smem =* HI *(src) << ASM* *Xmem =* HI *(src) << SHIFT1* *Smem =* HI *(src) << SHIFT*	**Store Accumulator High Into Memory** Store the 16 MSBs of the source accumulator in data memory (*Smem* or *Xmem*).
Smem = src *Smem = src << ASM* *Xmem = src << SHIFT1* *Smem = src << SHIFT*	**Store Accumulator Low Into Memory** Store the 16 LSBs of the source accumulator in data memory (*Smem* or *Xmem*).
MMR = src *mmr(MMR) = src*	**Store Accumulator Low Into Memory-Mapped Register** Store the 16 LSBs of the source accumulator into the memory-mapped register. The upper nine bits of the effective address are cleared to 0, regardless of the current value of DP or the upper nine bits of AR*X*.
MMR = #lk *mmr(MMR) = #lk*	**Store Immediate Value Into Memory-Mapped Register** Store the value into the specified memory-mapped register or any memory location on data page 0 without modifying the DP field in status register ST0.

Table 6–47. Conditional Store Instructions

Syntax	Description
cmps *(src,Smem)*	**Compare Select Max and Store** Compare the values located in the 16 MSBs and the 16 LSBs of the source accumulator (considered 2s-complement values). Store the maximum value in the memory location. Shift TRN left one bit.
if *(cond) Xmem* = HI *(src) << ASM*	**Store Accumulator Conditionally** If the condition is met, store the source accumulator (A or B) left-shifted as defined in *Xmem*. If the condition is not true, execute a read and a write at the *Xmem* address with the same value. Regardless of the condition, *Xmem* is always read and updated according to its modifier.
if *(cond) Xmem = BRC*	**Store Block Repeat Counter Conditionally** If the condition is true, store the value of the BRC in *Xmem.* If the condition is false, execute a read and a write at the *Xmem* address with the same value. Regardless of the condition, *Xmem* is always updated and read according to its modifier.
if *(cond) Xmem =* T	**Store TREG Conditionally** If the condition is true, store the value of TREG in *Xmem*. Regardless of the condition, *Xmem* is always read and updated according to its modifier. If the condition is true, execute a read and a write at the *Xmem* address with the same value.

Table 6–48. Parallel Load and Store Instructions

Syntax	Description
dst = *Xmem* [<< 16] || *dst* = *dst* + T**Ymem* *dst* = *Xmem* [<< 16] || *dst* += T**Ymem* *dst* = *Xmem* [<< 16] || *dst* = rnd *(dst* + T**Ymem)*	**Multiply/Accumulate With/Without Rounding and Parallel Load** Multiply the value *Ymem* by the value of TREG and add the result of the multiplication to the accumulator that is not the destination accumulator (*dst2*). At the same time, load the destination accumulator high with value *Xmem*.
dst = *Xmem* [<< 16] || *dst* = *dst* – T**Ymem* *dst* = *Xmem* [<< 16] || *dst* –= T**Ymem* *dst* = *Xmem* [<< 16] || *dst* = rnd *(dst* – T**Ymem)*	**Multiply/Subtract With/Without Rounding and Parallel Load** Multiply the value *Ymem* by the value of TREG and subtract the result of the multiplication from the accumulator that is not the destination accumulator (*dst2*). At the same time, load the destination accumulator high with the value *Xmem*.
Ymem = HI*(src)* [<< *ASM*] *|| dst = Xmem* << 16 *Ymem* = HI*(src)* [<<*ASM*] *||* T *= Xmem*	**Store Accumulator With Parallel Load** Store the value of the source accumulator shifted as defined by the ASM field of ST1 in the memory location *Ymem*. At the same time, load the value *Xmem* into the destination accumulator or TREG.

Table 6–49. Parallel Store and Multiply Instructions

Syntax	Description
Ymem = HI *(src)* [<< *ASM*] *|| dst = dst +* T ** Xmem* *Ymem* = HI *(src)* [<< *ASM*] *|| dst +=* T ** Xmem* *Ymem* = HI *(src)* [<< *ASM*] *|| dst =* rnd *(dst +* T ** Xmem)*	**Store Accumulator and Parallel Multiply/Accumulate With/ Without Rounding** Store the value of the source accumulator high shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, multiply the value *Xmem* by the value of TREG, add the result to the value of the destination accumulator, round if indicated, and store the final result in the destination accumulator.
Ymem = HI *(src) [<< ASM]* *|| dst = dst –* T ** Xmem* *Ymem* = HI *(src) [<< ASM]* *|| dst –=* T ** Xmem* *Ymem* = HI *(src) [<< ASM]* *|| dst =* rnd *(dst –* T ** Xmem)*	**Store Accumulator and Parallel Multiply/Subtract With/ Without Rounding** Store the value of the source accumulator high shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, multiply the value *Xmem* by the value of TREG, subtract the result from the value of the destination accumulator, round if indicated, and store the result in the destination accumulator.
Ymem = HI *(src) [<<ASM]* *|| dst =* T ** Xmem*	**Store Accumulator With Parallel Multiply** Store the value of the source accumulator high shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, multiply the value of TREG by the 16-bit dual addressed mode value *Xmem*, and store the result in the destination accumulator.

Table 6–50. Parallel Store and Add/Subtract Instructions

Syntax	Description
Ymem = HI *(src)* [<< *ASM*] *|| dst = dst +Xmem* << 16	**Store Accumulator With Parallel Add** Store the value of the source accumulator shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, add the value of the accumulator that is not the destination accumulator to the 16-bit left-shifted data memory location *Xmem*, and store the result in the destination accumulator.
Ymem = HI *(src) [<< ASM]* *|| dst = Xmem* << 16 *– dst*	**Store Accumulator With Parallel Subtract** Store the source accumulator high shifted as defined by the ASM field of ST1 in the memory location *Ymem.* At the same time, left shift the value *Xmem* 16 bits, subtract the value of the accumulator that is not the destination accumulator, and store the result in the destination accumulator.

Table 6–51. Miscellaneous Load-Type and Store-Type Instructions

Syntax	Description
Ymem = *Xmem*	**Move Data From Data Memory to Data Memory With X, Y Addressing** Move the value of the data memory location addressed by *Xmem* to the data memory location addressed by *Ymem*.
data *(dmad)* = *Smem*	**Move Data From Data Memory to Data Memory With Destination Addressing** Move the value of the addressed data memory location (*Smem*) to a data memory location addressed by *dmad.*
MMR = data *(dmad)* *mmr(MMR)* = data *(dmad)*	**Move Data From Data Memory to Memory-Mapped Register** Move the value of the data memory location (*dmad*) to a memory-mapped register (*MMR*).
prog *(pmad)* = *Smem*	**Move Data From Data Memory to Program Memory** Move a value (*Smem*) to a program memory location addressed by value *dmad*.
Smem = data *(dmad)*	**Move Data From Data Memory to Data Memory With Source Addressing** Move the value of the data memory location *dmad* to the value *Smem*.
data *(dmad)* = *MMR* data *(dmad)* = *mmr(MMR)*	**Move Data From Memory-Mapped Register to Data Memory** Move the value found in the memory-mapped register *MMR* to the value *dmad*.
MMRy = *MMRx* (0 <= x,y <= 8) *mmr(MMRy)* = *mmr(MMRx)*	**Move Data From Memory-Mapped Register to Memory-Mapped Register** Move the value of the memory-mapped register *MMRx* to the memory-mapped register *MMRy*.
Smem = prog *(pmad)*	**Move Data From Program Memory to Data Memory** Move the value found in the program memory address *pmad* to a data memory location addressed by *Smem*.
Smem = port (*PA)*	**Read Data From Port** Read a value from an external I/O *port* (*PA*) into the specified data memory location (*Smem*).

Table 6–51. Miscellaneous Load-Type and Store-Type Instructions (Continued)

Syntax	Description
port *(PA) = Smem*	**Write Data to Port** Write a value to an external I/O *port* (*PA*) from the specified data memory location (*Smem*).
Smem = prog *(A)*	**Read Program Memory Addressed by Accumulator A** Transfer a 16-bit word from a program memory location to a data memory location (*Smem*). The program memory address is specified by the 16 LSBs of accumulator A.
prog *(A) = Smem*	**Write Data Memory Addressed by Accumulator A** Transfer a 16-bit word from a data memory location (*Smem*) to a program memory location specified by the 16 LSBs of accumulator A.

Chapter 7

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

Topic **Page**

7.1 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. In this chapter, we use the terms *macro parameters* and *substitution symbols* interchangeably.

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 7.2, *Defining Macros*, for more information.
- Macros can 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. See Section 7.4, *Macro Libraries*, on page 7-14 for more information.

Step 2: **Call the macro.** After defining a macro, you call it by using the macro name as an opcode in the source program. This is referred to as a *macro call*.

Step 3: **Expand the macro.** The assembler expands your macros when the source program calls them. During expansion, the assembler passes arguments by variable to the macro parameters, replaces the macro call statement with the macro definition, and assembles the source code. By default, the macro expansions are printed in the listing file. You can turn off expansion listing by using the .mnolist directive. See Section 7.8, *Formatting the Output Listing*, on page 7-21 for more information.

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.

7.2 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, in an .include/.copy file, or in a macro library. For more information about macro libraries, see Section 7.4, *Macro Libraries*, on page 7-14.

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 7.9, *Using Recursive and Nested Macros*, on page 7-22.

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

macname	**.macro** [*parameter*$_1$] [**,** ... **,** *parameter*$_n$]
	model statements or macro directives
	[.mexit]
	.endm

macname	names the macro. You must place the name in the source statement's label field. Only the first 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	identifies the source statement as the first line of a macro definition. You must place .macro in the op-code field.
[*parameters*]	are optional substitution symbols that appear as operands for the .macro directive. Parameters are discussed in Section 7.3, *Macro Parameters/Substitution Symbols*, on page 7-6.
model statements	are instructions or assembler directives that are executed each time the macro is called.
macro directives	are used to control macro expansion.

[.mexit] functions as a goto .endm statement. The .mexit directive is useful when error testing confirms that macro expansion will fail.

.endm terminates the macro definition.

To include comments with your macro definition that do not appear in the macro expansion, precede your comments with an exclamation point. To include comments that do appear in the macro expansion, use an asterisk or semicolon. For more information about macro comments, see Section 7.7, *Producing Messages in Macros*, on page 7-19.

Example 7–1 shows the definition, call, and expansion of a macro.

Example 7–1. Macro Definition, Call, and Expansion

(a) Mnemonic example

```
       1              *
       2
       3              *        add3
       4              *
       5              *        ADDRP = P1 + P2 + P3
       6
       7              add3     .macro P1, P2, P3, ADDRP
       8
       9                       LD      P1, A
      10                       ADD     P2, A
      11                       ADD     P3, A
      12                       STL     A, ADDRP
      13                       .endm
      14
      15
      16                       .global abc, def, ghi, adr
      17
      18 000000      add3 abc, def, ghi, adr
 1
 1        000000 1000!          LD      abc, A
 1        000001 0000!          ADD     def, A
 1        000002 0000!          ADD     ghi, A
 1        000003 8000!          STL     A, adr
```

(b) Algebraic example

```
       1              *
       2
       3              *        add3
       4              *
       5              *        ADDRP = P1 + P2 + P3
       6
       7              add3     .macro P1, P2, P3, ADDRP
       8
       9                       A = P1
      10                       A = A + P2
      11                       A = A + P3
      12                       ADDRP = A
      13                       .endm
      14
      15
      16                       .global abc, def, ghi, adr
      17
      18 000000       add3 abc, def, ghi, adr
1
1        000000 1000!          A = @abc
1        000001 0000!          A = A + @def
1        000002 0000!          A = A + @ghi
1        000003 8000!          @adr = A
```

7.3 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.

7.3.1 Substitution Symbols

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.

Valid substitution symbols may be 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 (including substitution symbols defined with the .var directive) per macro. For more information about the .var directive, see subsection 7.3.7, *Substitution Symbols as Local Variables in Macros*, on page 7-13.

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 substitution symbol with its corresponding character string, then translates the source code into object code.

Example 7–2 shows the expansion of a macro with varying numbers of arguments.

Example 7–2. 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              Parms  100,label,x,y
;         a = 100                          ;  a = 100
;         b = label                        ;  b = label
;         c = ” ”                          ;  c = x,y

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

       Parms  ”””string”””,x,y
;         a = ”string”
;         b = x
;         c = y
```

7.3.2 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 7–3 shows character strings being assigned to substitution symbols.

Example 7–3. The .asg Directive

```
.asg  AR0,FP              ;  frame pointer
.asg  *AR1+,Ind           ;  indirect addressing
.asg  *AR1+0b,Rc_Prop     ;  reverse carry propagation
.asg  ”””string”””,strng   ;  string
.asg  ”a,b,c”,parms       ;  parameters
```

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 7–4 shows arithmetic being performed on substitution symbols.

Example 7–4. The .eval Directive

```
.asg   1,counter
.loop  100
.word  counter
.eval  counter + 1,counter
.endloop
```

In Example 7–4 the .asg directive could be replaced with the .eval directive 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 assigns the character string equivalent to a substitution symbol.

7.3.3 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 to these functions are substitution symbols or character-string constants.

In Table 7–1 function definitions, *a* and *b* are parameters that represent substitution symbols or character string constants. The term *string* refers to the string value of the parameter.

Table 7–1. Functions and Return Values

Function	Return Value
$symlen*(a)*	length of string *a*
$symcmp*(a,b)*	< 0 if *a* < *b* 0 if *a* = *b* > 0 if *a* > *b*
$firstch*(a,ch)*	index of the first occurrence of character constant *ch* in string *a*
$lastch*(a,ch)*	index of the last occurrence of character constant *ch* in string *a*
$isdefed*(a)*	1 if string *a* is defined in the symbol table 0 if string *a* is not defined in the symbol table
$ismember*(a,b)*	top member of list *b* is assigned to string *a* 0 if *b* is a null string
$iscons*(a)*	1 if string *a* is a binary constant 2 if string *a* is an octal constant 3 if string *a* is a hexadecimal constant 4 if string *a* is a character constant 5 if string *a* is a decimal constant
$isname*(a*)	1 if string *a* is a valid symbol name 0 if string *a* is not a valid symbol name
$isreg*(a)*†	1 if string *a* is a valid predefined register name 0 if string *a* is not a valid predefined register name

† For more information about predefined register names, see Section 3.8, *Symbols*, on page 3-16.

Example 7–5 shows built-in substitution symbol functions.

Example 7–5. Using Built-In Substitution Symbol Functions

```
.asg   label, ADDR                ; ADDR = label
.if    ($symcmp(ADDR,"label") = 0); evaluates to true
SUB    ADDR, A
.endif
.asg   "x,y,z" , list             ; list = x,y,z
.if    ($ismember(ADDR,list))     ; addr = x, list = y,z
SUB    ADDRA, A                   ; sub x
.endif
```

7.3.4 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 7–6, 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 7–6. 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"
      add      x, A


*    add      x, A   ; recursive expansion
```

7.3.5 Forced Substitution

In some cases, substitution symbols are not recognizable to the assembler. The forced substitution operator, which is a set of colons, 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 7–7 shows how the forced substitution operator is used.

Example 7–7. Using the Forced Substitution Operator

```
force  .macro x
       .asg      0,x
       .loop  8
AUX:x: .set   x
       .eval  x+1,x
       .endloop
       .endm
```

The force macro would generate the following source code:

```
AUX0   .set   0
AUX1   .set   1
   .
   .
   .
AUX7   .set   7
```

7.3.6 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 expression*$_1$, *well-defined expression*$_2$):

  In this method, expression$_1$ represents the substring's starting position, and expression$_2$ 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 7–8 and Example 7–9 show built-in substitution symbol functions used with subscripted substitution symbols.

Example 7–8. Using Subscripted Substitution Symbols to Redefine an Instruction

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

        ADDX     #100          ;macro call
        ADDX     *AR1          ;macro call
```

In Example 7–8, subscripted substitution symbols redefine the add instruction so that it handles short immediates.

Example 7–9. 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
          .if       $symcmp(strg1,TMP) = 0
          .eval     i,pos
          .break
          .else
          .eval     i + 1,i
          .endif
          .endloop
          .endm

          .asg      0,pos
          .asg      "ar1 ar2 ar3 ar4",regs
          substr    1,"ar2",regs,pos
          .word     pos
```

In Example 7–9, 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.

7.3.7 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 sym_1 [,sym_2] ... [,sym_n]

The .var directive is used in Example 7–8 and Example 7–9 on page 7-13.

7.4 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.

The syntax is:

.mlib *macro library filename*

When the assembler encounters the .mlib directive, it opens the library 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. You can control the listing of library entry expansions with the .mlist directive. For more information about the .mlist directive, see Section 7.8, *Formatting the Output Listing*, on page 7-21. Only macros that are actually called from the library are extracted, and they are extracted only once. For more information about the .mlib directive, see page 4-59.

You can create a macro library with the archiver by simply including the desired files in an archive. A macro library is no different from any other archive, except that the assembler expects the macro library to contain macro definitions. The assembler expects *only* macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable results.

7.5 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, and they 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 4-50.

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 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 4-58.

The **.break** directive and its expression are optional. 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 7–10, Example 7–11, and Example 7–12 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 7–10. 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 7–11. 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 7–12. Built-In Substitution Symbol Functions Used in a Conditional Assembly Code Block

```
       .fcnolist
*
*Double Add or Subtract
*
DBL    .macro ABC, ADDR, src      ; add or subtract double

       .if $symcmp(ABC,"+")
       dadd   ADDR, src           ; add double

       .elseif $symcmp(ABC,"–")
       dsub   ADDR, src           ; subtract double

       .else
       .emsg  "Incorrect Operator Parameter"

       .endif

       .endm

*Macro Call
       DBL  –, OPZ, ALP
```

For more information about conditional assembly directives, see Section 4.8, *Conditional Assembly Directives*, on page 4-20.

7.6 Using Labels in Macros

All labels in an assembly language program must be unique, including labels in macros. If a macro is expanded more than once, its labels are defined more than once, which *is illegal*. The macro language provides a method of defining labels in macros so that the labels are unique. Follow the label with a question mark, and the assembler replaces the question mark with a unique number. When the macro 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 macro definition. You cannot declare this label as global.

The maximum label length is shortened to allow for the unique suffix. 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.

The syntax for a unique label is:

label?

Example 7–13 shows unique label generation in a macro.

Example 7–13. Unique Labels in a Macro

(a) Mnemonic example

```
        1              ; define macro
        2              MIN      .macro AVAR, BVAR ; find minimum
        3
        4                       LD       AVAR, A
        5                       SUB      #BVAR, A
        6                       BC       M1?, ALT
        7                       LD       #BVAR, A
        8                       B        M2?
        9              M1?      LD       AVAR, A
       10              M2?
       11                      .endm
       12
       13              ; call macro
       14 000000               MIN 50, 100
1
1         000000 1032           LD       50, A
1         000001 F010           SUB      #100, A
          000002 0064
1         000003 F843           BC       M1?, ALT
          000004 0008'
1         000005 E864           LD       #100, A
1         000006 F073           B        M2?
          000007 0009'
1         000008 1032   M1?      LD       50, A
1         000009        M2?
```

Example 7–13. Unique Labels in a Macro (Continued)

(b) Algebraic example

```
        1              ; define macro
        2              MIN      .macro AVAR, BVAR ; find minimum
        3
        4                       A = AVAR
        5                       A = A – #BVAR
        6                       if (ALT) goto M1?
        7                       A = #BVAR
        8                       goto M2?
        9              M1?      A = AVAR
       10              M2?
       11                       .endm
       12
       13              ; call macro
       14 000000               MIN 50, 100
1
1         000000 1032           A = @50
1         000001 F010           A = A – #100
          000002 0064
1         000003 F843           if (A:T) goto M1$1$
          000004 0008'
1         000005 E864           A = #100
1         000006 F073           goto M2$1$
          000007 0009'
1         000008 1032   M1$1$   A = @50
1         000009        M2$1$
```

7.7 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 7–14 shows user messages in macros and macro comments that will not appear in the macro expansion.

Example 7–14. Producing Messages in a Macro

```
        1            testparam   .macro x,y
        2
        3                .if ($symlen(x) == 0)
        4                    .emsg ”ERROR –– Missing Parameter”
        5                    .mexit
        6                .elseif ($symlen(y) == 0)
        7                    .emsg ”ERROR == Missing Parameter”
        8                    .mexit
        9                .else
       10                    ld y, A
       11                    ld x, B
       12                    add A, B
       13                .endif
       14                .endm
       15
       16 0000          testparam 1,2
1
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          0000 1002        ld 2, A
1          0001 1101        ld 1, B
1          0002 F500        add A, B
1                       .endif
       17
       18 0003          testparam
1
1                       .if ($symlen(x) == 0)
1                           .emsg ”ERROR –– Missing Parameter”
  ***** USER ERROR ***** – : ERROR –– Missing Parameter
1                           .mexit

 1 Error, No Warnings
```

7.8 Formatting the Output Listing

Macros, substitution symbols, and conditional assembly directives may hide information. You may need to see this hidden information, so the macro language supports an expanded listing capability.

By default, the assembler shows macro expansions and false conditional blocks in the 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, .flist 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 the following directives in the listing file: .asg, .eval, .var, .sslist, .mlist, .fclist, .ssnolist, .mnolist, .fcnolist, .emsg, .wmsg, .mmsg, .length, .width, and .break.

  For directive listing, .drlist is the default.

7.9 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 7–15 shows nested macros. Note that 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 7–15. 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 7–16 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 7–16. Using Recursive Macros

(a) Mnemonic example

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

       ST     #1, loc
       .else
       ST     #N, loc   ; n >= 2 so, store n at loc
                        ; decrement n, and do the
       .eval  N – 1, N  ;      factorial of n – 1
       fact1            ; call fact with current
                        ;                environment
       .endif

       .endm

fact1  .macro

       .if    N > 1
       LD     loc, T    ; multiply present factorial
       MPY    #N, A     ;      by present position
       STL    A, loc    ; save result
       .eval  N – 1, N  ; decrement position
       fact1            ; recursive call
       .endif

       .endm
```

Example 7–16. Using Recursive Macros (Continued)

(b) Algebraic example

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

       @AR0 = #1
       .else
       @AR0 = #N       ; n >= 2 so, store n at loc
                       ; decrement n, and do the
       .eval N – 1, N  ;      factorial of n – 1
       fact1           ; call fact1 with current
                       ;                 environment
       .endif

       .endm

fact1  .macro

       .if N > 1
       T = @AR0        ; multiply present factorial
       A = T * #N      ;      by present position
       @AR0 = A        ; save result
       .eval N – 1, N  ; decrement position
       fact1           ; recursive call
       .endif

       .endm
```

7.10 Macro Directives Summary

Table 7–2. Creating Macros

Mnemonic and Syntax	Description
macname **.macro** [*parameter*$_1$]...[*parameter*$_n$]	Define macro.
.mlib *filename*	Identify library containing macro definitions.
.mexit	Go to .endm.
.endm	End macro definition.

Table 7–3. Manipulating Substitution Symbols

Mnemonic and Syntax	Description
.asg [**"**]*character string*[**"**], *substitution symbol*	Assign character string to substitution symbol.
.eval *well-defined expression, substitution symbol*	Perform arithmetic on numeric substitution symbols.
.var *substitution symbol*$_1$...[*substitution symbol*$_n$]	Define local macro symbols.

Table 7–4. Conditional Assembly

Mnemonic and Syntax	Description
.if *well-defined expression*	Begin conditional assembly.
.elseif *well-defined expression*	Optional conditional assembly block
.else	Optional conditional assembly block
.endif	End conditional assembly.
.loop [*well-defined expression*]	Begin repeatable block assembly.
.break [*well-defined expression*]	Optional repeatable block assembly.
.endloop	End repeatable block assembly.

Table 7–5. Producing Assembly-Time Messages

Mnemonic and Syntax	Description
.emsg	Send error message to standard output.
.wmsg	Send warning message to standard output.
.mmsg	Send warning or assembly-time message to standard output.

Table 7–6. Formatting the Listing

Mnemonic and Syntax	Description
.fclist	Allow false conditional code block listing (default).
.fcnolist	Inhibit false conditional code block listing.
.mlist	Allow macro listings (default).
.mnolist	Inhibit macro listings.
.sslist	Allow expanded substitution symbol listing.
.ssnolist	Inhibit expanded substitution symbol listing (default).

Chapter 8

Archiver Description

The TMS320C54x archiver combines several individual files into a single archive file. For example, you can collect several macros into a macro library. The assembler will search the library and use 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 will include in the library the members that resolve external references during the link.

Topic **Page**

8.1 Archiver Overview

The TMS320C54x archiver lets you combine several individual files into a single file called an archive or a library. Each file within the archive is called a member. Once you have created an archive, you can use the archiver to add, delete, or extract members.

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.

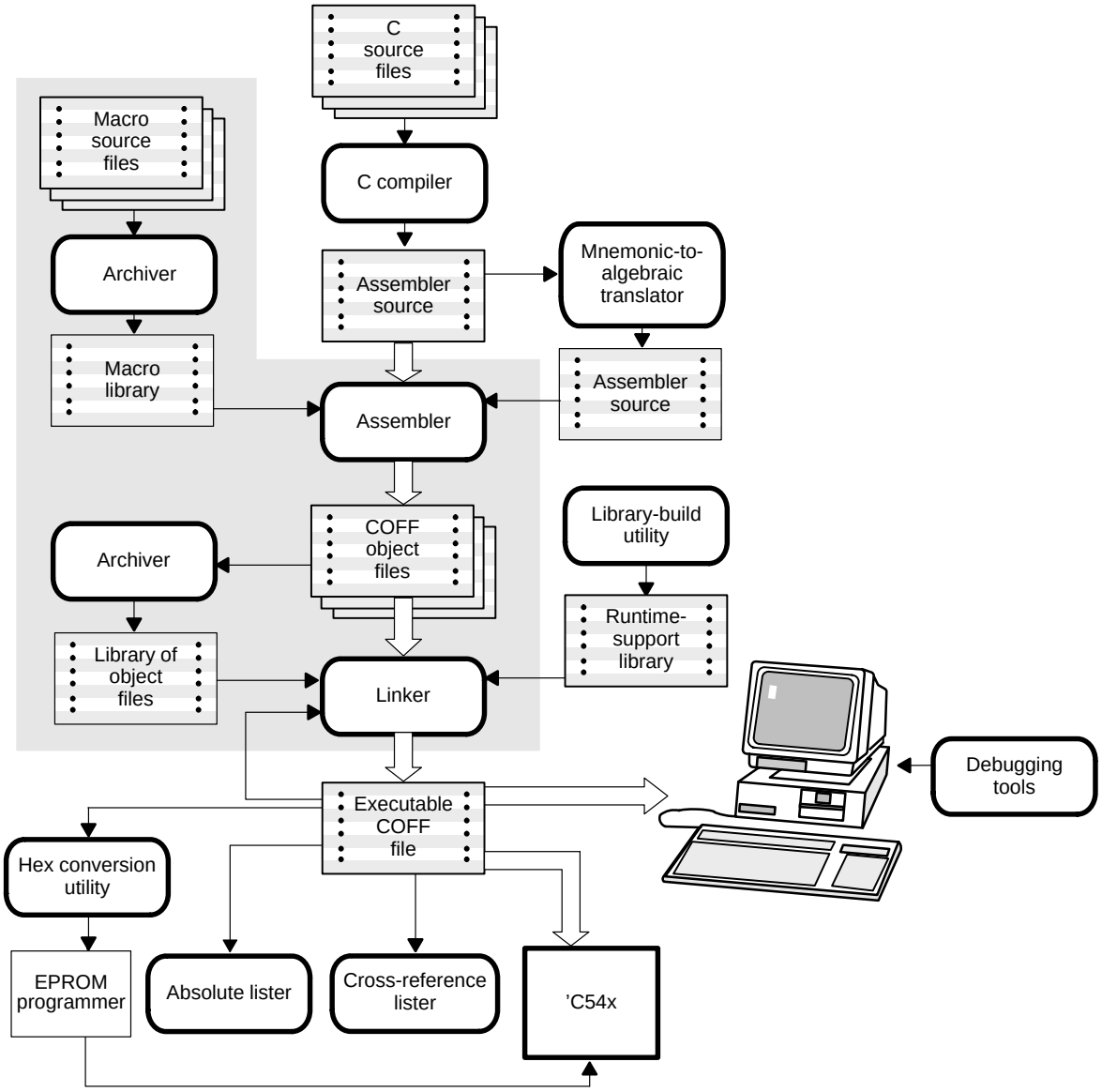
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The linker will search the library and include 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. The .mlib assembler directive lets you specify the name of a macro library; during the assembly process, the assembler will search the specified library for the macros that you call. Chapter 7, *Macro Language*, discusses macros and macro libraries in detail.

8.2 Archiver Development Flow

Figure 8–1 shows the archiver's role in the assembly language development process. Both the assembler and the linker accept libraries as input.

Figure 8–1. Archiver Development Flow



8.3 Invoking the Archiver

To invoke the archiver, enter:

ar500 [–]*command*[*option*] *libname* [*filename*$_1$ ... *filename*$_n$]

ar500 is the command that invokes the archiver.

command tells the archiver how to manipulate the library members. 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. Valid archiver commands are:

- **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 don't 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 don't specify any filenames, the archiver lists all the members in the specified library.
- **x** extracts the specified files. If you don't 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 *doesn't* remove it from the library.

option tells the archiver how to function. Specify as many of the following options as you want:

–e tells the archiver not to use the default extension .obj for member names. This allows the use of filenames without extensions.

–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 constituent members.

libname names an archive library. If you don't specify an extension for *libname,* the archiver uses the default extension *.lib*.

filename names individual member files that are associated with the library. If you don't specify an extension for a *filename,* the archiver uses the default extension *.obj*.

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, and the library contains more than one member with the specified name, then the archiver deletes, replaces, or extracts the first member with that name.

8.4 Archiver Examples

The following are some archiver examples:

- **Example 1**

  This example creates a library called function.lib that contains the files sine.obj, cos.obj, and flt.obj.

```
ar500 –a function sine cos flt
TMS320C54x Archiver              Version x.xx
Copyright (c) 1993–1996    Texas Instruments Incorporated
   ==>    new archive 'function.lib'
   ==>    building archive 'function.lib'
```

  Because Example 1 and 2 use the default extensions (.lib for the library and .obj for the members), it is not necessary to specify extensions in these examples.

- **Example 2**

  You can print a table of contents of function.lib with the –t option:

```
ar500 –t function
TMS320C54x Archiver              Version x.xx
Copyright (c) 1993–1996    Texas Instruments Incorporated
       FILE NAME    SIZE   DATE
-----------------   -----  ----------------------
       sine.obj         248    Mon Nov 19 01:25:44 1993
       cos.obj          248    Mon Nov 19 01:25:44 1993
       flt.obj          248    Mon Nov 19 01:25:44 1993
```

- **Example 3**

  You can explicitly specify extensions if you don't want the archiver to use the default extensions; for example:

```
ar500 –av function.fn sine.asm cos.asm flt.asm
TMS320C54x Archiver              Version x.xx
Copyright (c) 1993–1996    Texas Instruments Incorporated
   ==>    add 'sine.asm'
   ==>    add 'cos.asm'
   ==>    add 'flt.asm'
   ==>    building archive 'function.fn'
```

  This creates a library called function.fn that contains the files sine.asm, cos.asm, and flt.asm. (–v is the verbose option.)

❑ **Example 4**

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

```
ar500 –as function tan.obj arctan.obj area.obj
TMS320C54x Archiver            Version x.xx
Copyright (c) 1993–1996    Texas Instruments Incorporated
   ==>    symbol defined: 'K2'
   ==>    symbol defined: 'Rossignol'
   ==>    building archive 'function.lib'
```

Because this example doesn't specify an extension for the libname, the archiver adds the files to the library called function.lib. If function.lib didn't exist, the archiver would create it. (The –s option tells the archiver to list the global symbols that are defined in the library.)

❑ **Example 5**

If you want to modify a library member, you can extract it, edit it, and replace it. In this example, assume there's a library named macros.lib that contains the members push.asm, pop.asm, and swap.asm.

```
ar500 –x macros push.asm
```

The archiver makes a copy of push.asm and places it in the current directory, but it doesn't 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:

```
ar500 –r macros push.asm
```

Chapter 9

Linker Description

The TMS320C54x linker creates executable modules by combining COFF object files. 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**

9.1 Linker Overview

The TMS320C54x linker allows you to configure system memory by allocating output sections efficiently into the memory map. As the linker combines object files, it preforms 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.

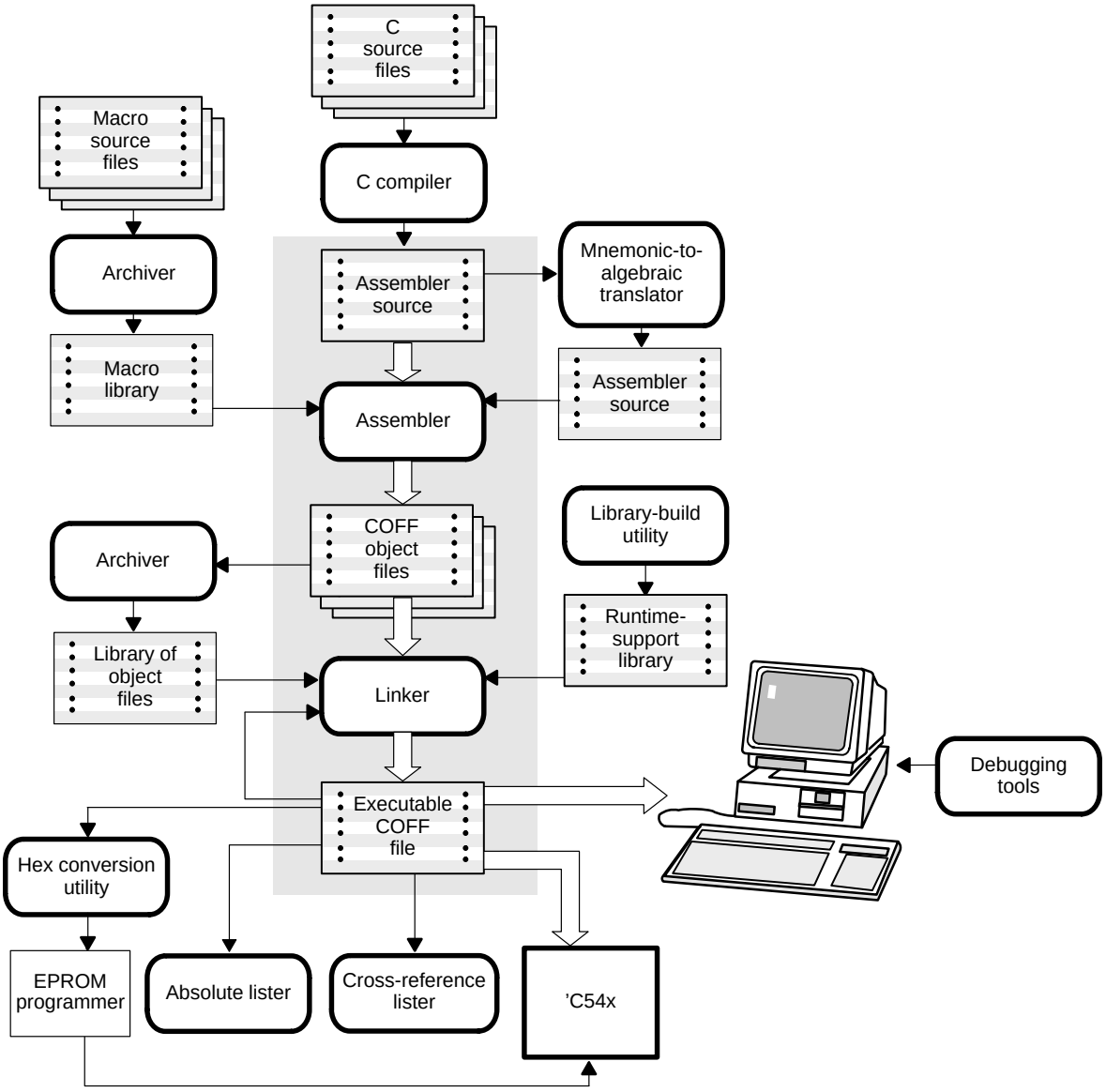
The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow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

- Allocate sections into specific areas of memory.
- Combine object file sections.
- Define or redefine global symbols at link time.

9.2 Linker Development Flow

Figure 9–1 illustrates the linker's role in the assembly language 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 TMS320C54x device.

Figure 9–1. Linker Development Flow



9.3 Invoking the Linker

The general syntax for invoking the linker is:

lnk500 [–*options*] *filename*$_1$. ... *filename*$_n$

lnk500 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 9.4, *Linker Options*, on page 9-6.)

filenames 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*.

There are three 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.

  ```
  lnk500 file1.obj file2.obj –o link.out
  ```

- Enter the **lnk500** command with no filenames and no options; the linker prompts for them:

  ```
  Command files :
  Object files [.obj] :
  Output file [a.out] :
  Options :
  ```

  - For *command files,* enter one or more command filenames.
  - For *object files,* enter one or more object filenames. The default extension is *.obj.* Separate the filenames with spaces or commas; if the last character is a comma, the linker prompts for an additional line of object filenames.
  - The *output file* is the name of the linker output module. This overrides any –o options entered with any of the other prompts. If there are no –o options and you do not answer this prompt, the linker creates an object file with a default filename of *a.out*.
  - The *options* prompt is for additional options, although you can also enter them in a command file. Enter them with hyphens, just as you would on the command line.

- Put filenames and options in a linker command file. For example, assume that the file linker.cmd contains the following lines:

```
–o link.out
file1.obj
file2.obj
```

  Now you can invoke the linker from the command line; specify the command filename as an input file:

```
lnk500 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:

```
lnk500 –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.

9.4 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 (–). The order in which options are specified is unimportant, except for the –l (lowercase L) and –i options. Options may be separated from arguments (if they have them) by an optional space. The following summarize the linker options:

–a	Produce an absolute, executable module. This is the default; if neither –a nor –r is specified, the linker acts as if –a were specified.
–ar	Produce a relocatable, executable object module.
–b	Disable merge of symbolic debugging information.
–c	Use linking conventions defined by the ROM autoinitialization model of the TMS320C54x C compiler.
–cr	Use linking conventions defined by the RAM autoinitialization model of the TMS320C54x C compiler.
–e *global_symbol*	Define a *global_symbol* that specifies the primary entry point for the output module.
–f *fill_value*	Set the default fill value for holes within output sections; *fill_value* is a 16-bit constant.
–g *global_symbol*	Make a *global_symbol* static (overrides –h).
–h	Make all global symbols static.
–heap *size*	Set heap size (for the dynamic memory allocation in C) to *size* words and define a global symbol that specifies the heap size. Default = 1K words.
–i *dir*	Alter the library-search algorithm to look in *dir* before looking in the default location. This option must appear before the –l option. The directory or filename must follow operating system conventions.
–k	Ignore alignment flags in input sections.
–l *filename*	Name an archive library file as linker input; *filename* is an archive library name. This option must appear after the –i option. The directory or filename must follow operating system conventions.

–m *filename*	Produce a map or listing of the input and output sections, including holes, and place the listing in *filename.* The directory or filename must follow operating system conventions. The directory or filename must follow operating system conventions.
–n	Ignore all fill specifications in memory directives.
–o *filename*	Name the executable output module. The default filename is a.out. The directory or filename must follow operating system conventions.
–q	Request a quiet run (suppress the banner).
–r	Produce a relocatable output module.
–s	Strip symbol table information and line number entries from the output module.
–stack *size*	Set C system stack size to *size* words and define a global symbol that specifies the stack size. The default size is 1K words.
–u *symbol*	Place an unresolved external *symbol* into the output module's symbol table.
–v*n*	Specify the output COFF format. The default format is COFF2.
–w	Displays a message when an undefined output section is created.
–x	Force rereading of libraries. Resolves back references.

9.4.1 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. If neither –a nor –r is specified, the linker acts as if –a is specified by default.

❑ **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 (subsection 9.14.4, *Symbols Defined by the Linker*, on page 9-58 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:

```
lnk500 –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 may contain unresolved references, but these references do not prevent creation of an output module.

The following example links file1.obj and file2.obj and creates a relocatable output module called a.out:

```
lnk500 -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 9.18, *Linker Example*, on page 9-69.

❑ Producing an Executable Relocatable Output Module (–ar)

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.

The following example links file1.obj and file2.obj and creates an executable, relocatable output module called xr.out:

```
lnk500 -ar file1.obj file2.obj –o xr.out
```

You can string the options together (lnk500 –ar) or enter them separately (lnk500 –a –r).

❑ Relocating or Relinking an Absolute Output Module

The linker issues a warning message (but continues executing) when it encounters a file that contains no relocation or symbol table information. 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).

9.4.2 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 will 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 do not 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.

9.4.3 C Language Options (–c and –cr Options)

The –c and –cr options cause the linker to use linking conventions that are required by the C compiler.

- The **–c** option tells the linker to use the ROM autoinitialization model.
- The **–cr** option tells the linker to use the RAM autoinitialization model.

For more information about linking C code, see Section 9.17, *Linking C Code*, on page 9-65 and subsection 9.17.5, *The –c and –cr Linker Options*, on page 9-68.

9.4.4 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 must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four possible 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 appear as an external symbol in one of the input files.

- The value of symbol _c_int00 (if present). _c_int00 *must* be the entry point if you are linking code produced by the C compiler.

- The value of symbol _main (if present).

- Zero (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.

```
lnk500 -e begin file1.obj file2.obj
```

9.4.5 Set Default Fill Value (–f *cc* 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 argument *cc* is a 16-bit constant (up to four 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 ABCD.

```
lnk500 -f 0ABCDh file1.obj file2.obj
```

9.4.6 Make All Global Symbols Static (–h and –g *global_symbol* Options)

The **–h** option makes all symbols defined with the .global assembler directive static. This effectively hides the symbols, because static symbols are not visible to externally linked modules. This allows external symbols with the same name (in different files) to be treated as unique. The **–g** option retains *global_symbol* as a global even if the –h option is used. The –h option will not modify symbols that were made global with the –g linker option.

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 that b1.obj, b2.obj, and b3.obj are related and reference a global variable GLOB. Also assume that d1.obj, d2.obj, and d3.obj are related and reference a separate global variable GLOB. By using the –h option and partial linking, you can link the related files without conflict.

```
lnk16 –h –r b1.obj b2.obj b3.obj –o bpart.out
lnk16 –h –r d1.obj d2.obj d3.obj –o dpart.out
```

The –h option guarantees that bpart.out and dpart.out do not have global symbols and therefore, that two distinct versions of GLOB exist. The –r option is used to allow bpart.out and dpart.out to retain their relocation entries. These two partially linked files can then be linked together safely with the following command:

```
lnk16 bpart.out dpart.out –o system.out
```

9.4.7 Define Heap Size (–heap *constant* Option)

The C compiler uses an uninitialized section called .sysmem for the C runtime memory pool used by malloc( ). You can set the size of this memory pool at link time by using the **–heap** option. Specify the size as a constant immediately after the option:

```
lnk500 –heap 0x0800 /* defines a 2k heap (.sysmem section)*/
```

The linker creates the .sysmem section only if there is a .sysmem section in one of the input files.

The linker also creates a global symbol __SYSMEM_SIZE and assigns it a value equal to the size of the heap. The default size is 1K words.

For more information about linking C code, see Section 9.17, *Linking C Code*, on page 9-65.

9.4.8 Alter the Library Search Algorithm (–i *dir* Option/C_DIR)

Usually, when you want to specify a library as linker input, you simply enter the library name as you would any other input 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:

```
lnk500 file1.obj object.lib
```

If you want to use a library that is not in the current directory, use the –l (lower-case L) linker option. The syntax for this option is;

–l *filename*

The *filename* is the name of an archive library; the space between –l and the filename is optional.

You can augment the linker's directory search algorithm by using the –i linker option or the environment variable. The linker searches for object libraries in the following order:

1) It searches directories named with the –i linker option.

2) It searches directories named with the environment variable C_DIR.

3) If C_DIR is not set, it searches directories named with the assembler's environment variable, A_DIR.

4) It searches the current directory.

9.4.8.1 Name an Alternate Library Directory (–i Option)

The –i option names an alternate directory that contains object libraries. The syntax for this option is:

–i *dir*

The *dir* names a directory that contains object libraries; the space between –i and the directory name is optional.

When the linker is searching for object libraries named with the –l option, it searches through directories named with –i first. Each –i option specifies only one directory, but you can use several –i options per invocation. When you use the –i option to name an alternate directory, it must precede the –l option 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	Invocation Command
DOS or OS/2	\ld and \ld2	`lnk500 f1.obj f2.obj –i\ld –i\ld2 –lr.lib –llib2.lib`
UNIX	/ld and /ld2	`lnk500 f1.obj f2.obj –i/ld –i/ld2 –lr.lib –llib2.lib`

9.4.8.2 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 commands for assigning the environment variable are:

Operating System	Enter
DOS or OS/2	`set C_DIR= pathname;another pathname ...`
UNIX	`setenv C_DIR "pathname;another pathname ..."`

The *pathnames* are directories that contain object libraries. Use the –l option on the command line or in a command file to tell the linker which libraries to search for.

For example, 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
DOS or OS/2	\ld and \ld2	`set C_DIR=\ld;\ld2` `lnk500 f1.obj f2.obj –l r.lib –l lib2.lib`
UNIX	/ld and /ld2	`setenv C_DIR "/ld ;/ld2"` `lnk500 f1.obj f2.obj –l r.lib –l lib2.lib`

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

Operating System	Enter
DOS or OS/2	`set C_DIR=`
UNIX	`unsetenv C_DIR`

The assembler uses an environment variable named **A_DIR** to name alternative directories that contain copy/include files or macro libraries. If C_DIR is not set, the linker will search for object libraries in the directories named with A_DIR. Section 9.6, *Object Libraries*, on page 9-24 contains more information about object libraries.

9.4.9 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 may also contain up to three tables:

- A table showing the new memory configuration if any nondefault memory is specified
- A table showing the linked addresses of each output section and the input sections that make up the output sections
- A table showing each external symbol and its address. This table has two columns: the left column contains the symbols sorted by name, and the right column contains the symbols sorted by address

This example links file1.obj and file2.obj and creates a map file called map.out:

```
lnk500 file1.obj file2.obj –m map.out
```

Example 9–14 on page 9-71 shows an example of a map file.

9.4.10 Ignore the Memory Directive Fill Specification (–n Option)

The **–n** option forces the linker to ignore any MEMORY directive fill specification. This option can be used in the development stage of a project to avoid generating large .out files, which can result from the use of MEMORY directive fill specifications.

9.4.11 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:

```
lnk500 –o run.out file1.obj file2.obj
```

9.4.12 Specify a Quiet Run (–q Option)

The **–q** option suppresses the linker's banner when –q is the first option on the command line or in a command file. This option is useful for batch operation.

9.4.13 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 must create the smallest possible output module.

This example links file1.obj and file2.obj and creates an output module, stripped of line numbers and symbol table information, named nosym.out:

```
lnk500 –o nosym.out –s file1.obj file2.obj
```

Using the –s option limits later use of a symbolic debugger and may prevent a file from being relinked.

9.4.14 Define Stack Size (–stack *constant* Option)

The TMS320C54x C compiler uses an uninitialized section, .stack, to allocate space for the runtime stack. You can set the size of the .stack section at link time with the **–stack** option. Specify the size as a constant immediately after the option:

```
lnk500 –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 will be 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 1K words.

9.4.15 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.

For example, suppose a library named rts.lib contains a member that defines the symbol symtab; none of the object files being linked reference symtab. However, suppose you plan to relink the output module, and you would like 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.

```
lnk500 –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.

9.4.16 Specify a COFF Format (–v Option)

The **–v** option specifies the format the linker will use to create the COFF object file. The COFF object file is the output of the linker. The format specifies how information in the object file is layed out.

The linker can read and write COFF0, COFF1, and COFF2 formats. By default, the linker creates COFF2 files. To create a different output format, use the –v option where *n* is 0 for COFF0 or 1 for COFF1.

For more information about COFF files see Chapter 2, *Introduction to Common Object File Format*, and Appendix A, *Common Object File Format*.

9.4.17 Display a Message for Output Section Information (–w Option)

The –w option displays additional messages pertaining to the creation of memory sections. Additional messages are displayed in the following circumstances:

- 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 when this has occurred.

  If this situation occurs and you use the –w option, the linker displays a message when it creates a new output section.

- If you do not use the –heap and –stack options, the linker creates the .sysmem and .stack, respectively, sections for you. Each section has a default size of 0x400 words. You might not have enough memory available for one or both of these sections. In this case, the linker issues an error message saying a section could not be allocated.

  If you use the –w option, the linker displays another message with more details, which includes the name of the directive to allocate the .sysmem or .stack section yourself.

For more information about the SECTIONS directive, see Section 9.8, *The SECTIONS Directive*, on page 9-30. For more information about the default actions of the linker, see Section 9.12, *Default Allocation Algorithm*, on page 9-51.

9.4.18 Exhaustively Read Libraries (–x Option)

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 will not be 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 the –x option 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:

```
lnk500 –la.lib –lb.lib –la.lib
```

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

```
lnk500 –x –la.lib –lb.lib
```

9.5 Linker Command Files

Linker command files allow you to put linking information in a file; this is useful when you invoke the linker often with the same information. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line.

Linker command files are ASCII files that contain one or more of the following:

- Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)
- Linker options, which can be used in the command file in the same manner that they are used on the command line
- The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration. The SECTIONS directive controls how sections are built and allocated.
- Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the **lnk500** command and follow it with the name of the command file:

lnk500 *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 it. 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 9–1 shows a sample linker command file called link.cmd. (Subsection 2.4.2, *Placing Sections in the Memory Map*, on page 2-14 contains another example of a linker command file.)

Example 9–1. 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 9–1 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:

```
lnk500 link.cmd
```

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

```
lnk500 –r link.cmd c.obj d.obj
```

The linker processes the command file as soon as it encounters it, 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:

```
lnk500 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 9–2 shows a sample command file that contains linker directives. (Linker directive formats are discussed in later sections.)

Example 9–2. Command File With Linker Directives

```
a.obj b.obj c.obj           /* Input filenames       */
–o prog.out –m prog.map     /* Options               */

MEMORY                      /* MEMORY directive      */
{
  RAM:  origin = 100h      length = 0100h
  ROM:  origin = 01000h    length = 0100h
}

SECTIONS                    /* SECTIONS directive    */
{
  .text:  > ROM
  .data:  > ROM
  .bss:   > RAM
}
```

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

9.5.2 Constants in Command Files

Constants can be specified 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 3-13) or the scheme used for integer constants in C syntax.

Examples:

	Decimal	Octal	Hexadecimal
Assembler Format:	32	40q	20h
C Format:	32	040	0x20

9.6 Object Libraries

An object library is a partitioned archive file that contains complete 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 8, *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, it is linked whether it is used or not; however, if that same function is placed in an archive library, it is included only if it is referenced.

The order in which libraries are specified is important because the linker includes only those members that resolve symbols that are undefined when 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. 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. Assume that:

- 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

For example, if you enter the following, the references are resolved as shown:

```
lnk500 f1.obj liba.lib f2.obj libc.lib
```

- Member 1 of liba.lib satisfies both references to clrscr because the library is searched and clrscr is defined before f2.obj references it.
- Member 0 of libc.lib satisfies the reference to origin.
- Member 3 of liba.lib satisfies the reference to fillclr.

As another example, if you enter the following, all the references to clrscr are satisfied by member 1 of libc.lib:

```
lnk500 f1.obj f2.obj libc.lib liba.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. The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
lnk500 –u rout1 libc.lib
```

If any member of libc.lib define rout1, the linker includes those members.

It is not possible to control the allocation of individual library members; members are allocated according to the SECTIONS directive default allocation algorithm.

Subsection 9.4.8, *Alter the Library Search Algorithm (–i dir Option/C_DIR)*, on page 9-13, describes methods for specifying directories that contain object libraries.

9.7 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 TMS320C54x 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.

Refer to Section 2.4, *How the Linker Handles Sections*, on page 2-12 for details on how the linker handles sections. Refer to Section 2.5, *Relocation*, on page 2-14 for information on the relocation of sections.

9.7.1 Default Memory Model

The assembler enables you to assemble code for the TMS320C54x device. The assembler inserts a field in the output file's header, identifying the 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 subsection 9.12.1, *Allocation Algorithm*, on page 9-51.

9.7.2 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 memory range has a *name,* a *starting address,* and a *length.*

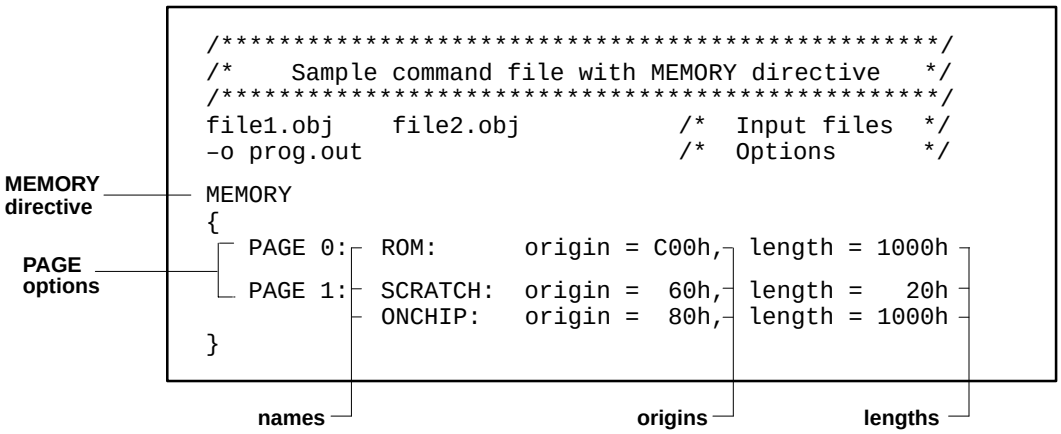
'C54x devices have separate memory spaces that occupy the same address ranges. In the default model, one space is dedicated to program areas, while a second is dedicated to data (the number of separate address spaces depends on the customized configuration of your chip. See the *TMS320C54x User's Guide* for more information.)

The linker allows you to configure these address spaces separately by using the MEMORY directive's PAGE option. In the default model, PAGE 0 refers to program memory, and PAGE 1 refers to data memory. The linker treats these two pages as completely separate memory spaces. The 'C54x supports as many as 255 PAGES, but the number available to you depends on the configuration you have chosen.

When you use the MEMORY directive, be sure to identify *all* the memory ranges that are available for object code. Memory defined by the MEMORY directive is *configured memory*; any memory that you do not explicitly account for with the MEMORY directive is *unconfigured memory*. The linker does not place any part of a program into unconfigured memory. You can represent non-existent 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 9–3 defines a system that has 4K words of ROM at address 0h in program memory, 32 words of RAM at address 60h in data memory, and 512 words at address 200h in data memory.

Example 9–3. The MEMORY Directive



You could then use the SECTIONS directive to tell the linker where to link the sections. For example, you could allocate the .text and .data sections into the memory area named ROM and allocate the .bss section into B2 or B0B1.

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control allocation of output sections. After you use the MEMORY directive to specify the target system's memory model, you can use the SECTIONS directive 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 ROM and allocate the .bss section into the area name ONCHIP.

The general syntax for the MEMORY directive is:

```
MEMORY
{
    PAGE 0 :  name 1 [(attr)]  : origin = constant ,  length = constant;
    PAGE n :  name n [(attr)]  : origin = constant ,  length = constant;
}
```

PAGE identifies a memory space. You can specify up to 255 pages, depending on your configuration; usually, PAGE 0 specifies program memory, and PAGE 1 specifies data memory. If you do not specify a PAGE, the linker acts as if you specified PAGE 0. Each PAGE represents a completely independent address space. Configured memory on PAGE 0 can overlap configured memory on PAGE 1.

name Names a memory range. A memory name may be one to eight 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. Memory ranges on separate pages can have the same name; within a page, however, all memory ranges must have unique names and must not overlap.

attr 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 include:

- **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 *origin*, *org*, or *o*. The value, specified in bytes, is a 16-bit constant and may be decimal, octal, or hexadecimal.

length Specifies the length of a memory range; enter as *length*, *len*, or *l*. The value, specified in bytes, is a 16-bit constant and may be decimal, octal, or hexadecimal.

fill Specifies a fill character for the memory range; enter as *fill* or *f*. Fills are optional. The value is a 2-byte integer constant and may be decimal, octal, or hexadecimal. The fill value will be used to fill areas of the memory range that are not allocated to a section.

Note: Filling Memory Ranges

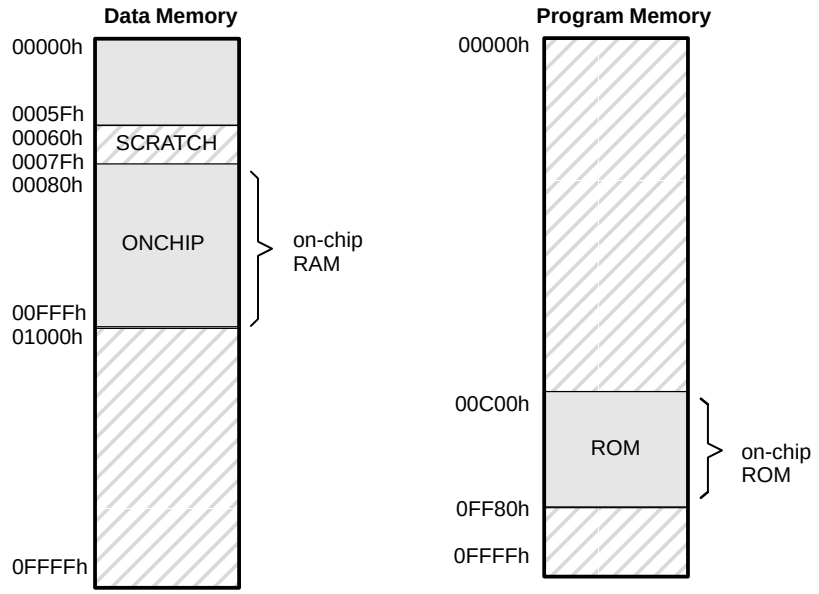
If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFh:

```
MEMORY
{
       RFILE (RW) : o = 02h, l = 0FEh, f = 0FFFFh
}
```

Figure 9–2 illustrates the memory map shown in Example 9–3.

Figure 9–2. Memory Map Defined in Example 9–3



9.8 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

Refer to Section 2.4, *How the Linker Handles Sections*, on page 2-12 for details on how the linker handles sections. Refer to Section 2.5, *Relocation*, on page 2-14 for information on the relocation of sections. Refer to subsection 2.3.4, *Subsections*, on page 2-8 for information on defining subsections; subsections allow you to manipulate sections with greater precision.

9.8.1 Default Configuration

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. Section 9.12, *Default Allocation Algorithm*, on page 9-51 describes this algorithm in detail.

9.8.2 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.) After the section name is a list of properties that define the section's contents and how the section is allocated. The properties may be separated by optional commas. Possible properties for a section are:

- **Load allocation**, which defines where in memory the section is to be loaded
  Syntax: **load =** *allocation* **or**
  allocation **or**
  > *allocation*

- **Run allocation**, which defines where in memory the section is to be run
  Syntax: **run =** *allocation* **or**
  run > *allocation*

- **Input sections**, which define the input sections that constitute the output section
  Syntax: { input_sections }

- **Section type**, which defines flags for special section types
  Syntax: **type = COPY** **or**
  type = DSECT **or**
  type = NOLOAD
  For more information on section types, see Section 9.13, *Special Section Types (DSECT, COPY, and NOLOAD)*, on page 9-54.

- **Fill value**, which defines the value used to fill uninitialized holes
  Syntax: **fill =** *value* **or**
  name: ... { ... } **=** *value*
  For more information on creating and filling holes, see Section 9.15, *Creating and Filling Holes*, on page 9-59.

Example 9–4 shows a SECTIONS directive in a sample linker command file. Figure 9–3 shows how these sections are allocated in memory.

Example 9–4. The SECTIONS Directive

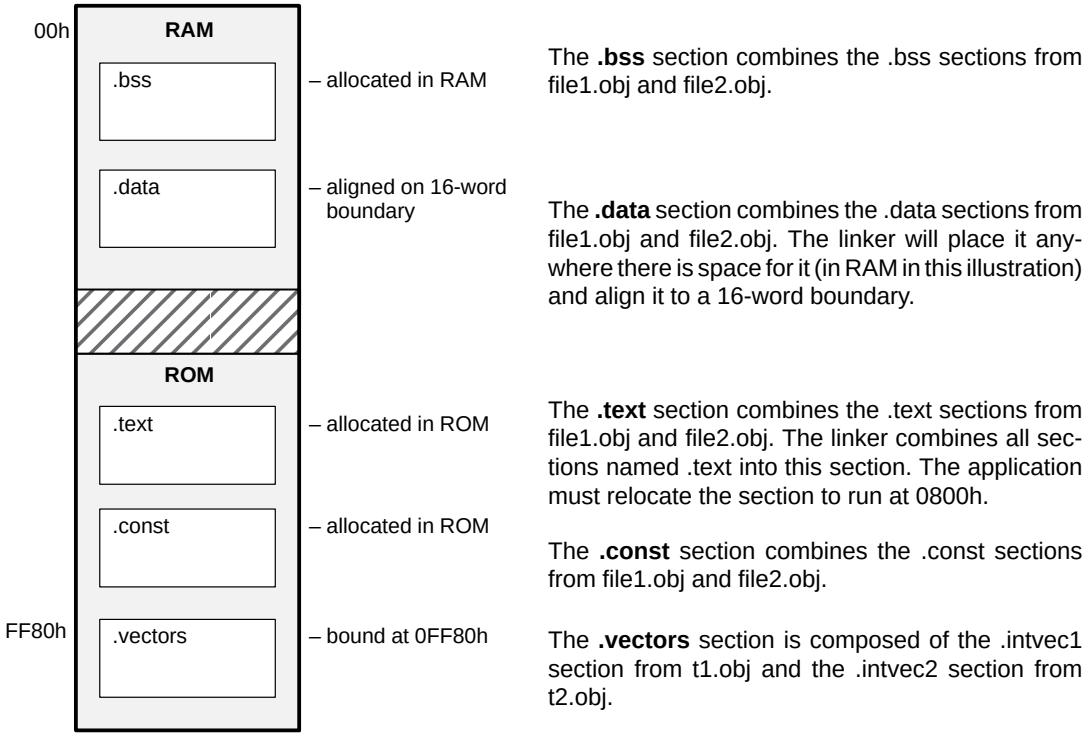
SECTIONS directive

section specifications

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

SECTIONS
{
   .text:     load = ROM, run = 800h
   .const:    load = ROM
   .bss:      load = RAM
   .vectors:  load = FF80h
      {
         t1.obj(.intvec1)
         t2.obj(.intvec2)
         endvec = .;
      }
   .data:     align = 16
}
```

Figure 9–3 shows the five output sections defined by the sections directive in Example 9–4: .vectors, .text, .const, .bss, and .data.

Figure 9–3. Section Allocation Defined by Example 9–4



9.8.3 Specifying the Address of Output Sections (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. In any case, 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 9.9, *Specifying a Section's Runtime Address*, on page 9-39.

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to allocate the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run allocation is separate, all parameters following the keyword LOAD apply to load allocation, and those following RUN apply to run allocation. Possible allocation parameters are:

Binding allocates a section at a specific address.

```
.text: load = 0x1000
```

Memory allocates the section into a range defined in the MEMORY directive with the specified name (like ROM) or attributes.

```
.text: load > ROM
```

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

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

To force the output section containing the assignment to also be aligned, assign . (dot) with an align expression. For example, the following will align bar.obj, and it will force outsect to align on a 0x40 boundary:

```
SECTIONS
{
   outsect:  {  bar.obj(.bss)
                . = align(0x40);
             }
}
```

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)
```

Page specifies the memory page to be used (see Section 9.11, *Overlay Pages*, on page 9-46).

```
.text: PAGE 0
```

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

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

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

```
.text: > ROM align 16 PAGE 2
```

Or, if you prefer, use parentheses for readability:

```
.text: load = (ROM align(16) page (2))
```

9.8.3.1 Binding

You can supply a specific starting address for an output section by following the section name with an address:

```
.text: 0x1000
```

This example specifies that the .text section must begin at location 1000h. The binding address must be a 16-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 and Alignment or Named Memory are Incompatible

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.

9.8.3.2 Named memory

You can allocate a section into a memory range that is defined by the MEMORY directive. This example names ranges and links sections into them:

```
MEMORY
{
   ROM (RIX)  :  origin = C00h,  length = 1000h
   RAM (RWIX) :  origin =  80h,  length = 1000h
}

SECTIONS
{
   .text  :             > ROM
   .data ALIGN(128) :   > RAM
   .bss   :             > RAM
```

In this example, the linker places .text into the area called ROM. The .data and .bss output sections are allocated into RAM. You can align a section within a named memory range; the .data section is aligned on a 128-word boundary within the RAM 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 ROM or RAM area because both areas have the X attribute. The .data section can also go into either ROM or RAM because both areas have the R and I attributes. The .bss output section, however, must go into the RAM area because only RAM 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 would start at address 0. If a section must start on a specific address, use binding instead of named memory.

9.8.3.3 Alignment and blocking

You can tell the linker to place an output section at an address that falls on an n-word boundary, where n is a power of 2. For example:

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

allocates .text so that it falls on a page boundary.

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 will begin on that boundary. As with alignment, n must be a power of 2. For example:

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

allocates .bss so that the section either is contained in a single 128K-word page or begins on a page.

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

9.8.3.4 Specifying input sections

An input section specification identifies the sections from input files that are combined to form an output section. The linker combines input sections by concatenating them in the order in which they are specified. The size of an output section is the sum of the sizes of the input sections that comprise it.

Example 9–5 shows the most common type of section specification; note that no input sections are listed.

Example 9–5. The Most Common Method of Specifying Section Contents

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

In Example 9–5 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 9–5 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 than the input sections' name.
- You want the linker to allocate the input sections *before* it processes additional input sections or commands within the braces.

The following example illustrates the two purposes above:

```
SECTIONS
{
   .text  :  {
                    abc.obj(xqt)
                       *(.text)
             }
   .data  :  {
                    *(.data)
                    fil.obj(table)
             }
}
```

In this example, the .text output section contains a named section xqt from file abc.obj, which is followed by all the .text input sections. The .data section contains all the .data input sections, followed by a named section table from the file fil.obj. This method includes all the unallocated sections. For example, if one of the .text input sections was already included in another output section when the linker encountered *(.text), the linker could not include that first .text input section in the second output section.

9.9 Specifying a Section's Runtime 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 a ROM-based system. The code must be loaded into ROM, but it would run faster in RAM.

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 = ROM, run = RAM
```

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

Refer to Section 2.6, *Runtime Relocation*, on page 2-16 for an overview on runtime relocation.

9.9.1 Specifying Load and Run Addresses

The load address determines where a loader will place 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 will load and run 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 subsection 9.10.1, *Overlaying Sections With the UNION Statement*, on page 9-43.)

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 may also specify run first, then load. Use parentheses to improve readability.

The examples below specify load and run addresses:

```
.data: load = ROM, align = 32, run = RAM
```

(align applies only to load)

```
.data: load = (ROM align 32), run = RAM
```

(identical to previous example)

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

(align 32 in RAM for run; align 16 anywhere for load)

9.9.2 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. The example below specifies load and run addresses for an uninitialized section:

```
.bss: load = 0x1000, run = RAM
```

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

```
.bss: load = RAM
.bss: run = RAM
.bss: > RAM
```

9.9.3 Referring to the Load Address by Using the .label Directive

Normally, any reference to a symbol in a section refers to its runtime address. However, it may be necessary at runtime 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 4-53.

Example 9–6 shows the use of the .label directive.

Example 9–6. Copying a Section From ROM to RAM

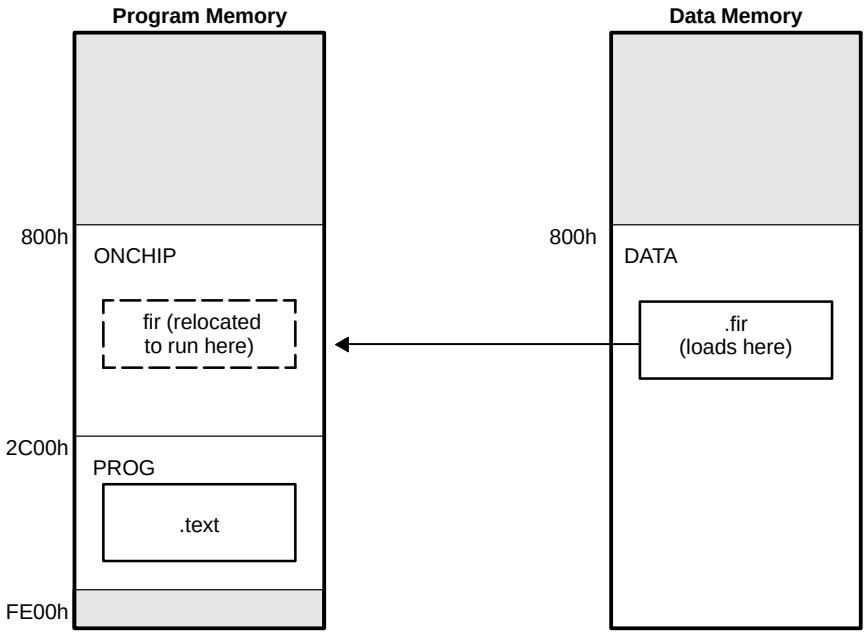
```
;----------------------------------------------------
;  define a section to be copied from ROM to RAM
;----------------------------------------------------
    .sect  ".fir"
    .label fir_src        ; load address of section
fir:                      ; run address of section
    <code here>           ; code for the section

    .label fir_end        ; load address of section end

;----------------------------------------------------
;  copy .fir section from ROM into RAM
;----------------------------------------------------
    .text

    STM   fir_src, AR1    ; get load address
    RPT   #(fir_end – fir_src – 1)
    MVDP  *AR1+, fir      ; copy address to program memory

;----------------------------------------------------
;  jump to section, now in RAM
;----------------------------------------------------
      CALL  fir
```

Linker Command File

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

MEMORY
{
   PAGE 0 :  ONCHIP :  origin = 0800h,  length = 02400h
   PAGE 0 :  PROG   :  origin = 02C00h, length = 0D200h
   PAGE 1 :  DATA   :  origin = 0800h,  length – 0F800h
}

SECTIONS
{
   .text: load = PROG PAGE 0
   .fir:  load = DATA PAGE 1, run ONCHIP PAGE 0
}
```

Figure 9–4 illustrates the runtime execution of this example.

Figure 9–4. Runtime Execution of Example 9–6



9.10 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.

9.10.1 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 on-chip RAM at various stages of execution. Or you may want several data objects that will not be 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 runtime address.

In Example 9–7, the .bss sections from file1.obj and file2.obj are allocated at the same address in RAM. 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 9–7. The UNION Statement

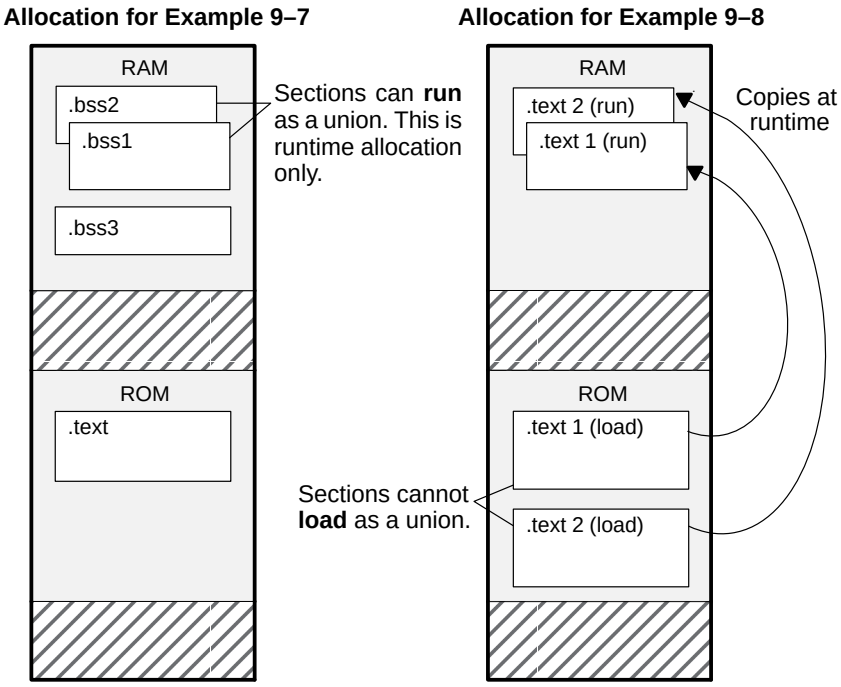
```
SECTIONS
{
   .text: load = ROM
   UNION: run = RAM
   {
      .bss1: { file1.obj(.bss) }
      .bss2: { file2.obj(.bss) }
   }
      .bss3: run = RAM { globals.obj(.bss) }
```

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 has raw data, such as .text), its load allocation *must* be separately specified. For example:

Example 9–8. Separate Load Addresses for UNION Sections

```
   UNION: run = RAM
   {
      .text1: load = ROM, { file1.obj(.text) }
      .text2: load = ROM, { file2.obj(.text) }
   }
```

Figure 9–5. Memory Allocation Shown in Example 9–7 and Example 9–8



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 fits in configured memory.

Uninitialized sections are not loaded and do not require load addresses.

The UNION statement applies only to allocation of run addresses, so it is redundant 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 are specified, the linker issues a warning and ignores the load address.

Note: UNION and Overlay Page Are Not the Same

The UNION capability and the *overlay page* capability (see Section 9.11, *Overlay Pages*, on page 9-46) may sound similar because they both deal with overlays. They are, in fact, quite different. UNION allows multiple sections to be overlaid *within the same memory space*. Overlay pages, on the other hand, define *multiple memory spaces*. It is possible to use the page facility to approximate the function of UNION, but this is cumbersome.

9.10.2 Grouping Output Sections Together

The SECTIONS directive has a GROUP option that 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 9–9. Allocate Sections Together

```
SECTIONS
{
   .text          /* Normal output section             */
   .bss           /* Normal output section             */
   GROUP 1000h :  /* 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 1000h. This means that .data is allocated at 1000h, 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.

9.11 Overlay Pages

Some target systems use a memory configuration in which all or part of the memory space is overlaid by shadow memory. This allows the system to map different banks of physical memory into and out of a single address range in response to hardware selection signals. In other words, multiple banks of physical memory overlay each other at one address range. You may want the linker to load various output sections into each of these banks or into banks that are not mapped at load time.

The linker supports this feature by providing *overlay pages*. Each page represents an address range that must be configured separately with the MEMORY directive. You can then use the SECTIONS directive to specify the sections to be mapped into various pages.

9.11.1 Using the MEMORY Directive to Define Overlay Pages

To the linker, each overlay page represents a completely separate memory comprising the full 16-bit range of addressable locations. This allows you to link two or more sections at the same (or overlapping) addresses if they are on different pages.

Pages are numbered sequentially, beginning with 0. If you do not use the PAGE option, the linker allocates initialized sections into PAGE 0 (program memory) and uninitialized sections into PAGE 1 (data memory).

For example, assume that your system can select between two banks of physical memory for data memory space: address range A00h to FFFFh for PAGE 1 and 0A00h to 2BFF for PAGE 2. Although only one bank can be selected at a time, you can initialize each bank with different data. This is how you use the MEMORY directive to obtain this configuration:

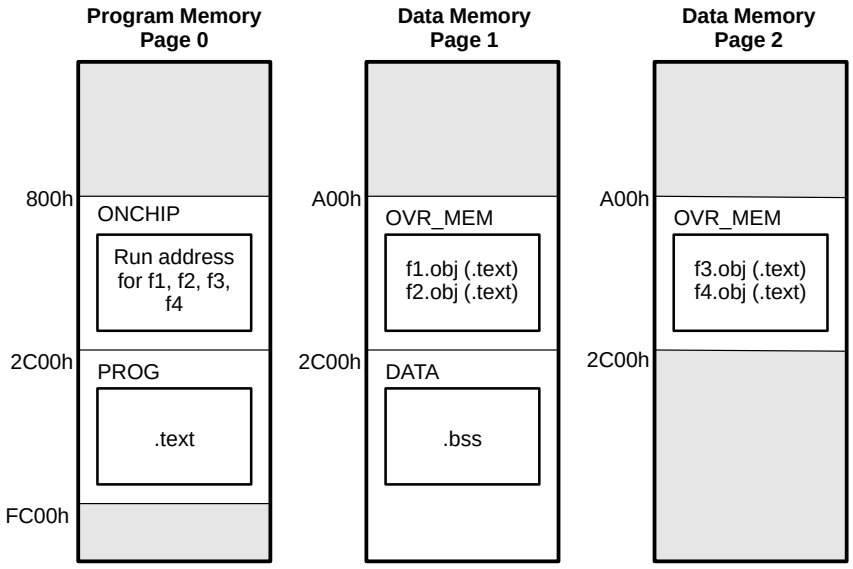
Example 9–10. Memory Directive With Overlay Pages

```
MEMORY
{
   PAGE 0 :  ONCHIP   : origin = 0800h,   length = 0240h
          :  PROG     : origin = 02C00h,  length = 0D200h
   PAGE 1 :  OVR_MEM  : origin = 0A00h,   length = 02200h
          :  DATA     : origin = 02C00h,  length = 0D400h
   PAGE 2 :  OVR_MEM  : origin = 0A00h,   length = 02200h
}
```

Example 9–10 defines three separate address spaces. PAGE 0 defines an area of on-chip program memory and the rest of program memory space. PAGE 1 defines the first overlay memory area and the rest of data memory space. PAGE 2 defines another area of overlay memory for data space. Both OVR_MEM ranges cover the same address range. This is possible because each range is on a different page and therefore represents a different memory space.

Figure 9–6 shows overlay pages defined by the MEMORY directive in Example 9–10 and the SECTIONS directive in Example 9–11.

Figure 9–6. Overlay Pages Defined by Example 9–10 and Example 9–11



9.11.2 Using Overlay Pages With the SECTIONS Directive

Assume that you are using the MEMORY directive as shown in Example 9–10. Further assume that your code consists of, besides the usual sections, four modules of code that you want to load in data memory space but that you intend to run in the on-chip RAM in program memory space. Example 9–11 shows how to use the SECTIONS directive overlays accordingly.

Example 9–11. SECTIONS Directive Definition for Overlays in Figure 9–6

```
SECTIONS
{
   UNION :  run = ONCHIP
   {
      S1 :  load = OVR_MEM PAGE 1
      {
         s1_load = 0A00h;
         s1_start = .;
         f1.obj (.text)
         f2.obj (.text)
         s1_length = . – s1_start;
      }
      S2 :  load = OVR_MEM PAGE 2
      {
         s2_load = 0A00h;
         s2_start = .;
         f3.obj (.text)
         f4.obj (.text)
         s2_length = . – s2_start;
      }
   }

   .text: load = PROG PAGE 0
   .data: load = PROG PAGE 0
   .bss : load = DATA PAGE 1
}\
```

The four modules of code are f1, f2, f3, and f4. The modules f1 and f2 are combined into output section S1, and f3 and f4 are combined into output section S2. The PAGE specifications for S1 and S2 tell the linker to link these sections into the corresponding pages. As a result, they are both linked to load address A00h, but in different memory spaces. When the program is loaded, a loader can configure hardware so that each section is loaded into the appropriate memory bank.

Output sections S1 and S2 are placed in a union that has a run address in on-chip RAM. The application must move these sections at runtime before executing them. You can use the symbols s1_load and s1_length to move section S1, and s2_load and s2_length to move section S2. The special symbol "." refers to the current run address, not the current load address.

Within a page, you can bind output sections or use named memory areas in the usual way. In Example 9–11, S1 could have been allocated:

```
S1 : load = 01200h, page = 1  { . . . }
```

This binds S1 at address 1200h in page 1. You can also use page as a qualifier on the address. For example:

```
S1 : load = (01200h PAGE 1)  { . . . }
```

If you do not specify any binding or named memory range for the section, the linker allocates the section into the page wherever it can (just as it normally does with a single memory space). For example, S2 could also be specified as:

```
S2 : PAGE 2  { . . . }
```

Because OVR_MEM is the only memory on page 2, it is not necessary (but acceptable) to specify = OVR_MEM for the section.

9.11.3 Page Definition Syntax

To specify overlay pages as illustrated in Example 9–10 and Example 9–11, use the following syntax for the MEMORY directive:

MEMORY
{
PAGE 0 : *name 1* [*(attr)*] **: origin =** *constant* **, length =** *constant;*
PAGE *n* **:** *name n* [*(attr)*] **: origin =** *constant* **, length =** *constant;*
}

Each page is introduced by the keyword PAGE and a page number, followed by a colon and a list of memory ranges the page contains. **Bold** portions must be entered as shown. Memory ranges are specified in the normal way. You can define up to 255 overlay pages.

Because each page represents a completely independent address space, memory ranges on different pages can have the same name. Configured memory on any page can overlap configured memory on any other page. Within a single page, however, all memory ranges must have unique names and must not overlap.

Memory ranges listed outside the scope of a PAGE specification default to PAGE 0. Consider the following example:

```
MEMORY
{           ROM   : org = 0h     len = 1000h
            EPROM : org = 1000h  len = 1000h
            RAM   : org = 2000h  len = 0E000h
   PAGE1:   XROM  : org = 0h     len = 1000h
            XRAM  : org = 2000h  len = 0E000h
}
```

The memory ranges ROM, EPROM, and RAM are all on PAGE 0 (since no page is specified). XROM and XRAM are on PAGE 1. Note that XROM on PAGE 1 overlays ROM on PAGE 0, and XRAM on PAGE 1 overlays RAM on PAGE 0.

In the output link map (obtained with the –m linker option), the listing of the memory model is keyed by pages. This provides an easy method of verifying that you specified the memory model correctly. Also, the listing of output sections has a PAGE column that identifies the memory space into which each section will be loaded.

9.12 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. Subsections 9.12.1, *Allocation Algorithm*, and 9.12.2, *General Rules for Output Sections*, describe default allocation.

9.12.1 Allocation Algorithm

If you do not use the MEMORY and SECTIONS directives, the linker allocates output sections as though the following definitions are specified.

Example 9–12. Default Allocation for TMS320C54x Devices

```
MEMORY
{
   PAGE 0: PROG: origin = 0x0080     length = 0xFF00
   PAGE 1: DATA: origin = 0x0080     length = 0xFF80
}
SECTIONS
{
   .text:     PAGE = 0
   .data:     PAGE = 0
   .cinit:    PAGE = 0    ;cflag option only
   .bss:      PAGE = 1
}
```

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. The .text and .data sections are allocated into configured memory on PAGE 0, which is the program memory space. All .bss sections are combined to form a .bss output section. The .bss section is allocated into configured memory on PAGE 1, which is the data memory space.

If the input files contain initialized named sections, the linker allocates them into program memory following the .data section. If the input files contain uninitialized named sections, the linker allocates them into data memory following the .bss section. You can override this by specifying an explicit PAGE in the SECTIONS directive.

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 in subsection 9.12.2, *General Rules for Output Sections*.

9.12.2 General Rules for Output Sections

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

Rule 1 As the result of a SECTIONS directive definition.

Rule 2 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 (rule 1), this definition completely determines the section's contents. (See Section 9.8, *The SECTIONS Directive*, on page 9-30 for examples of how to define an output section's content.)

An output section can also be formed when input sections are not specified by a SECTIONS directive (rule 2). In this case, 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.

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.

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. This is the algorithm:

1) Output sections for which you have supplied a specific binding address are placed in memory at that address.

2) Output sections that are included in a specific, named memory range or that have memory attribute restrictions are 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.

Note: The PAGE Option

If you do not use the PAGE option to explicitly specify a memory space for an output section, the linker allocates the section into PAGE 0. This occurs even if PAGE 0 has no room and other pages do. To use a page other than PAGE 0, you must specify the page with the SECTIONS directive.

9.13 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 2000h    (DSECT)    : {f1.obj}
   sec2 4000h    (COPY)     : {f2.obj}
   sec3 6000h    (NOLOAD)   : {f3.obj}
}
```

- The DSECT type creates a dummy section with the following qualities:
  - 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 TMS320C54x C compiler has this attribute under the RAM 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 it, and it appears in the memory map listing.

9.14 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.

9.14.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the 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 subsection 9.14.3, *Assignment Expressions*. Assignment statements *must* terminate with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Therefore, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol cur_tab as the address of the current table. cur_tab 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 */
```

9.14.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the SPC during allocation. 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.

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, 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 will relocate 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 ".". Assignments to "." and holes are described in Section 9.15, *Creating and Filling Holes*, on page 9-59.

9.14.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in Table 9–1.
- 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 9–1 in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in Table 9–1, the linker also has an *align* operator that allows a symbol to be aligned on an *n*-word 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-word 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 9–1. Operators Used in Expressions (Precedence)

Symbols	Operators	Evaluation
+ – ~	Unary plus, minus, 1s complement	Right to left
* / %	Multiplication, division, modulo	Left to right
+ –	Addition, subtraction	Left to right
<< >>	Left shift, right shift	Left to right
< <= > >=	Less than, LT or equal, greater than, GT or equal	Left to right
!=, =[=]	Not equal to, equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right

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

9.14.4 Symbols Defined by the Linker

The linker automatically defines several symbols that a program can use at runtime to determine where a section is linked. These symbols are external, so they appear in the link map. They can be accessed in any assembly language module if they are declared with a .global directive. 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.)

9.14.5 Symbols Defined Only For C Support (–c or –cr Option)

__STACK_SIZE is assigned the size of the .stack section.

__SYSMEM_SIZE is assigned the size of the .sysmem section.

9.15 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.

9.15.1 Initialized and Uninitialized Sections

An output section contains *one* of the following:

- 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 and sections defined with the .usect directive 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 RAM for variables. In the object file, an uninitialized section has a normal section header and may have symbols defined in it; however, no memory image is stored in the section.

9.15.2 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 follow the first guideline above and 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 holes. 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 9.14, *Assigning Symbols at Link Time*, on page 9-55.

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-word 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-word 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 file3.obj .text to start on a 16-word boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, file3.obj .text will not be aligned either.

Note that the “.” symbol refers to the current run address, not the current load address, of the section.

Expressions that decrement “.” are illegal. For example, it is invalid to use the –= operator in an assignment to “.”. 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 (rule 1). 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.

9.15.3 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 16-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 16-bit constant:

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      file2.obj(.bss) = 00FFh   /* Fill this hole */
   }                            /* with 0FFh      */
}
```

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 = 0FF00h /* fills holes with 0FF00h  */
   {
      . += 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 –f. For example, suppose the command file link.cmd contains the following SECTIONS directive:

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

Now invoke the linker with the –f option:

```
lnk500 -f 0FFFFh link.cmd
```

This fills the hole with 0FFFFh.

4) If you do not invoke the linker with the –f option, 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.

9.15.4 Explicit Initialization of Uninitialized Sections

An uninitialized section becomes a hole only when it is combined with an initialized section. When uninitialized sections are combined with each other, the resulting output section remains uninitialized.

However, 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 = 1234h      /* Fills .bss with 1234h */
}
```

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.

9.16 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:

- ❏ Intermediate files *must* have relocation information. Use the –r option when you link the file the first time.
- ❏ 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.
- ❏ 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 subsection 9.4.6, *Make All Global Symbols Static (–h and –g global_symbol Options)*, on page 9-12.)
- ❏ If you are linking C code, don't use –c or –cr until the final link step. Every time you invoke the linker with the –c or –cr option the linker will attempt to create an entry point.

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.

```
lnk500 –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.

```
lnk500 –r –o tempout2 file2.com
```

file2.com contains:

```
SECTIONS
{
   ss2:   {
          g1.obj
          g2.obj
           .
           .
           .
          gn.obj
          }
}
```

Step 3: Link tempout1.out and tempout2.out:

```
lnk500 –m final.map –o final.out tempout1.out tempout2.out
```

9.17 Linking C Code

The TMS320C54x 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:

```
lnk500 –c –o prog.out prog1.obj prog2.obj ... rts.lib
```

The –c option tells the linker to use special conventions that are defined by the C environment. The archive library rts.lib contains C runtime-support functions.

For more information about C, including the runtime environment and runtime-support functions, see the *TMS320C54x Optimizing C Compiler User's Guide.*

9.17.1 Runtime Initialization

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

- Sets up the system stack
- Processes the runtime initialization table and autoinitializes global variables (in the ROM model)
- Disables interrupts and calls _main

The runtime-support object library, rts.lib, contains boot.obj. You can:

- Use the archiver to extract boot.obj from the library and then link the module in directly.
- Include rts.lib as an input file (the linker automatically extracts boot.obj when you use the –c or –cr option).

9.17.2 Object Libraries and Runtime Support

The *TMS320C54x Optimizing C Compiler User's Guide* describes additional runtime-support functions that are included in *rts.lib*. If your program uses any of these functions, you must link *rts.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.

9.17.3 Setting the Size of the Stack and Heap Sections

C uses two uninitialized sections called .sysmem and .stack for the memory pool used by the malloc( ) functions and the runtime stack, respectively. You can set the size of these by using the –heap option or –stack option and specifying the size of the section as a constant immediately after the option. The default size for both is 1K words.

For more information, see Section 9.4.7, *Define Heap Size (–heap constant Option)*, on page 9-12 and subsection 9.4.14, *Define Stack Size (–stack constant Option)*, on page 9-18.

9.17.4 Autoinitialization (ROM and RAM Models)

The C compiler produces tables of data for autoinitializing global variables. These are in a named section called *.cinit.* The initialization tables can be used in either of two ways:

- **RAM Model** (–cr option)

  Variables are initialized at *load time.* This enhances performance by reducing boot time and by saving memory used by the initialization tables. You must use a smart loader (i.e. one capable of initializing variables) to take advantage of the RAM model of autoinitialization.

  When you use –cr, the linker marks the .cinit section with a special attribute. This attribute tells the linker *not* to load the .cinit section into memory. The linker also sets the cinit symbol to –1; this tells the C boot routine that initialization tables are not present in memory. Thus, no runtime initialization is performed at boot time.

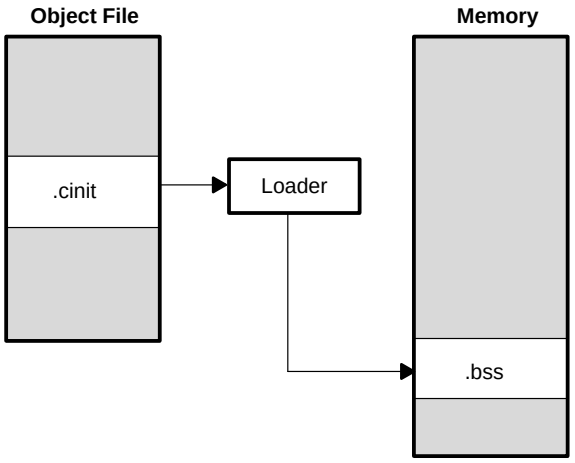
  When the program is loaded, the loader must be able to:

  - Detect the presence of the .cinit section in the object file
  - Detect the presence of the attribute that tells it not to copy the .cinit section
  - Understand the format of the initialization tables. (This format is described in the *TMS320C54x Optimizing C Compiler User's Guide.)*

  The loader then uses the initialization tables directly from the object file to initialize variables in .bss.

Figure 9–7 illustrates the RAM autoinitialization model.

Figure 9–7. RAM Model of Autoinitialization

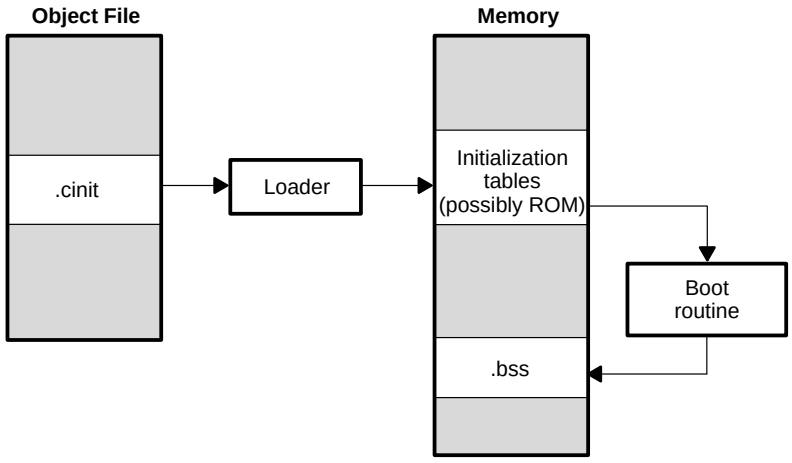


- **ROM Model** (–c option)

  Variables are initialized at *runtime.* The .cinit section is loaded into memory along with all the other sections. The linker defines a special symbol called cinit that points to the beginning of the tables in memory. When the program begins running, the C boot routine copies data from the tables into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program is started.

  Figure 9–8 illustrates the ROM autoinitialization model.

Figure 9–8. ROM Model of Autoinitialization



9.17.5 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. _c_int00 is the start of the C boot routine in boot.obj; referencing _c_int00 ensures that boot.obj is automatically linked in from the runtime-support library rts.lib.
- The .cinit output section is padded with a termination record to designate to the boot routine (ROM model) or the loader (RAM model) when to stop reading the initialization tables.
- In the ROM model (–c option), the linker defines the symbol cinit as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.
- In the RAM model (–cr option):
  - The linker sets the symbol cinit to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at runtime.
  - The STYP_COPY flag (0010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.

9.18 Linker Example

This example links three object files named demo.obj, fft.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
0080 to 7000	On-chip RAM_PG
C000 to FF80	On-chip ROM

Data Memory

Address Range	Contents
0080 to 0FFF	RAM block ONCHIP
0060 to FFFF	Mapped external addresses EXT

The output sections are constructed from the following input sections:

- Executable code, contained in the .text sections of demo.obj, fft.obj, and tables.obj must be linked into program ROM.
- Variables, contained in the var_defs section of demo.obj, must be linked into data memory in block ONCHIP.
- Tables of coefficients in the .data sections of demo.obj, tables.obj and fft.obj must be linked into RAM block ONCHIP in data memory. A hole is created with a length of 100 and a fill value of 07A1Ch. The remainder of block ONCHIP must be initialized to the value 07A1Ch.
- The .bss sections from demo.obj. tables.obj, and fft.obj, which contain variables, must be linked into block RAM_PG of program RAM. The unused part of this RAM must be initialized to 0FFFFh.
- The xy section from demo.obj, which contains buffers and variables, will have the default linking into block ONCHIP of data RAM, since it was not explicitly linked.

Example 9–13 shows the linker command file for this example. Example 9–14 shows the map file.

Example 9–13. Linker Command File, demo.cmd

```
/*************************************************************/
/***                 Specify Linker Options                ***/
/*************************************************************/
-e coeff                    /* Define the program entry point */
-o demo.out                 /* Name the output file           */
-m demo.map                 /* Create an output map           */

/*************************************************************/
/***                  Specify the Input Files              ***/
/*************************************************************/

demo.obj
fft.obj
tables.obj

/*************************************************************/
/***            Specify the Memory Configurations          ***/
/*************************************************************/

MEMORY
{
        PAGE 0: RAM_PG:  origin=00080h   length=06F80h
                   ROM:  origin=0C000h   length=03F80h

        PAGE 1: ONCHIP:  origin=00080h   length=0F7Fh
                   EXT:  origin=01000h   length=0EFFFh
}

/*************************************************************/
/***              Specify the Output Sections              ***/
/*************************************************************/

SECTIONS
{
        .text: load = ROM, page = 0    /*  link .text into ROM */

        var_defs: load = ONCHIP, page=1 /*     defs in RAM     */

        .data: fill = 07A1Ch, load=ONCHIP, page=1
         {
                tables.obj(.data) /* .data input */
                fft.obj(.data)    /* .data input */
                . = 100h;     /* create hole, fill with 07A1Ch */
         }                    /* and link with ONCHIP          */

        .bss: load=RAM_PG,page=0,fill=0FFFFh
                              /* Remaining .bss; fill and link */
}

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

Invoke the linker with the following command:

```
lnk500 demo.cmd
```

This creates the map file shown in Example 9–14 and an output file called demo.out that can be run on a TMS320C54x.

Example 9–14. Output Map File, demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: 0

MEMORY CONFIGURATION
           name      origin     length     attributes     fill
         --------   --------   ---------   ----------   --------
PAGE 0: RAM_PG     00000080   000006f80      RWIX
           ROM     0000c000   000003f80      RWIX
PAGE 1: ONCHIP     00000080   000000f7f      RWIX
           EXT     00001000   00000efff      RWIX

SECTION ALLOCATION MAP
 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text      0    0000c000    00000015
                0000c000    00000008     demo.obj (.text)
                0000c008    00000007     fft.obj (.text)
                0000c00f    00000006     tables.obj (.text)
var_defs   1    00000080    00000002
                00000080    00000002     demo.obj (var_defs)
.data      1    00000082    00000108
                00000082    00000000     tables.obj (.data)
                00000082    00000000     fft.obj (.data)
                00000082    00000100     --HOLE-- [fill = 7a1c]
                00000182    00000008     demo.obj (.data)
.bss       0    00000080    0000007b
                00000080    00000013     demo.obj (.bss) [fill=ffff]
                00000093    00000000     fft.obj (.bss)
                00000093    00000068     tables.obj (.bss) [fill=ffff]
xy         1    0000018a    00000014     UNINITIALIZED
                0000018a    00000014     demo.obj (xy)

GLOBAL SYMBOLS
address  name                             address  name
-------- ----                             -------- ----
00000080 .bss                             00000080 .bss
00000082 .data                            00000082 .data
0000c000 .text                            00000093 TEMP
00000097 ARRAY                            00000097 ARRAY
00000093 TEMP                             000000fb end
0000018a edata                            0000018a edata
000000fb end                              0000c000 .text
0000c015 etext                            0000c015 etext

[8 symbols]
```

Chapter 10

Absolute Lister Description

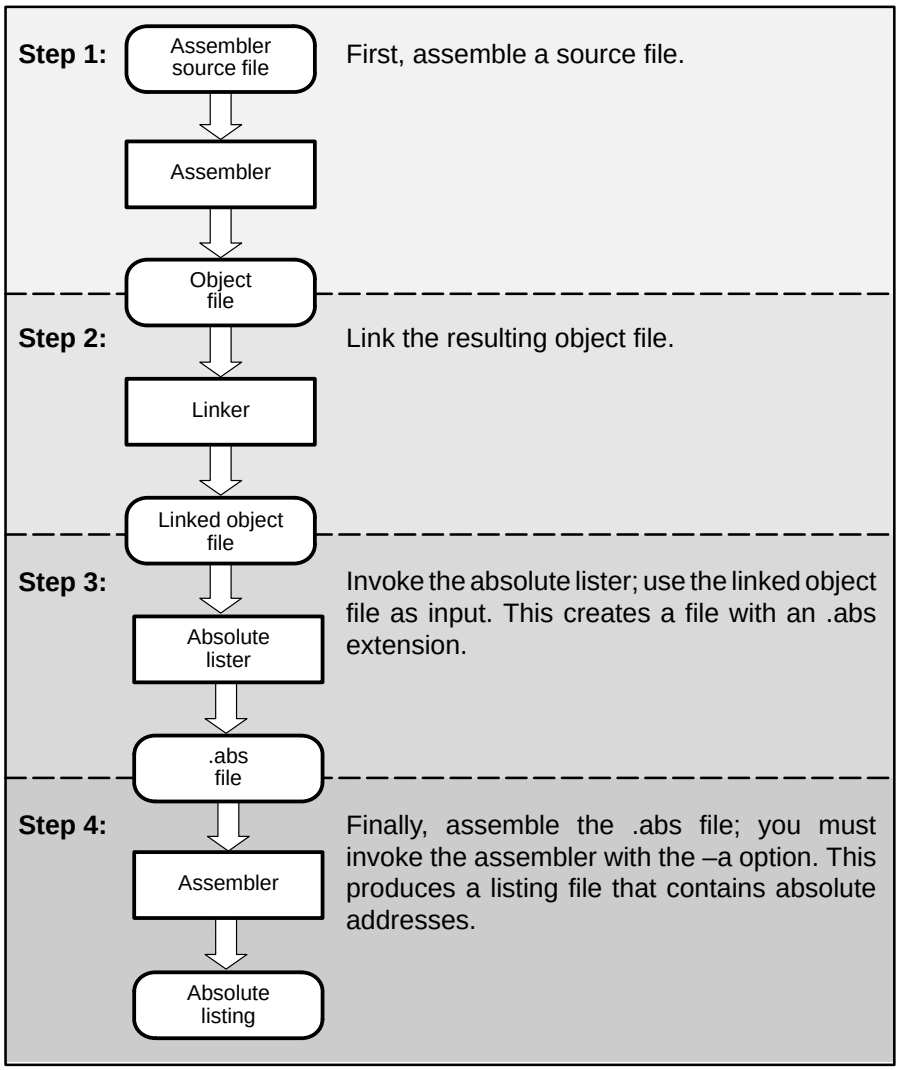
The absolute lister is a debugging tool that accepts linked object files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code. Manually, this could be a tedious process requiring many operations; however, the absolute lister utility performs these operations automatically.

Topic **Page**

10.1 Producing an Absolute Listing

Figure 10–1 illustrates the steps required to produce an absolute listing.

Figure 10–1. Absolute Lister Development Flow



10.2 Invoking the Absolute Lister

The syntax for invoking the absolute lister is:

abs500 [–*options*] *input file*

abs500 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 the default naming conventions for filename extensions on assembly files, C source files, and C header files. The three 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.

–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 will prompt 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:

```
asm500 –a filename.abs
```

The –e options affect both the interpretation of filenames on the command line and the names of the output files. They should always precede any filename on the command line.

The –e options are useful when the linked object file was created from C files compiled with the debugging option (–g compiler option). When the debugging option is set, the resulting linked object file contains the name of the source files used to build it. In this case, the absolute lister will not generate a corresponding .abs file for the C header files. Also, the .abs file corresponding to a C source file will use 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 assembly file hello.s. hello.csr also includes hello.hsr. Assuming the executable file created is called hello.out, the following command will generate the proper .abs file:

```
abs500 –ea s –c csr –eh hsr hello.out
```

An .abs file will not be created for hello.hsr (the header file), and hello.abs will include the assembly file hello.s, not the C source file hello.csr.

10.3 Absolute Lister Example

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

module1.asm

```
.text
.bss   array,100
.bss   dflag, 2
.copy  globals.def
ld     #offset, A
ld     dflag, A
```

module2.asm

```
.bss   offset, 2
.copy  globals.def
ld     #offset, A
ld     #array, A
```

globals.def

```
.global dflag
.global array
.global offset
```

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

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

```
asm500 module1
asm500 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 TMS320C54x 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
{
   PAGE 0:   ROM:  origin=2000h   length=2000h
   PAGE 1:   RAM:  origin=8000h   length=8000h
}

/************************************************/
/*         Specify the Output Sections          */
/************************************************/
SECTIONS
{
   .data:  >RAM
   .text:  >ROM
   .bss:   >RAM
}
```

Invoke the linker:

`lnk500 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:

`abs500 bttest.out`

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

module1.abs:

```
          .nolist
array     .setsym     08000h
dflag     .setsym     08064h
offset    .setsym     08066h
.data     .setsym     08000h
edata     .setsym     08000h
.text     .setsym     02000h
etext     .setsym     02007h
.bss      .setsym     08000h
end       .setsym     08068h
          .setsect    ".text",02000h
          .setsect    ".data",08000h
          .setsect    ".bss",08000h
          .list
          .text
          .copy       "module1.asm"
```

module2.abs:

```
          .nolist
array     .setsym     08000h
dflag     .setsym     08064h
offset    .setsym     08066h
.data     .setsym     08000h
edata     .setsym     08000h
.text     .setsym     02000h
etext     .setsym     02007h
.bss      .setsym     08000h
end       .setsym     08068h
          .setsect    ".text",02003h
          .setsect    ".data",08000h
          .setsect    ".bss",08066h
          .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 symbol *dflag*. The symbol dflag was defined in the file globals.def, which was included in module1.asm and module2.asm.
- They contain .setsect directives, which define the absolute addresses for sections.
- They contain .copy directives, which 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):

```
asm500  –a  module1.abs
asm500  –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 (see Figure 10–2) and module2.lst (see Figure 10–3).

Figure 10–2. module1.lst

```
TMS320C54x COFF Assembler       Version x.xx     Wed Oct 16 12:00:05 1996
 Copyright (c) 1993–1996    Texas Instruments Incorporated

module1.abs                                                          PAGE    1

      15 2000                 .text
      16                      .copy       ”module1.asm”
 A     1 2000               .text
 A     2 8000               .bss   array, 100
 A     3 8064               .bss   dflag, 2
 A     4                    .copy  globals.def
 B     1                    .global dflag
 B     2                    .global array
 B     3                    .global offset
 A     5 2000 F020          ld     #offset, A
         2001 8066!
 A     6 2002 1064–         ld     dflag, A

 No Errors, No Warnings
```

Figure 10–3. module2.lst

```
TMS320C54x COFF Assembler       Version x.xx     Wed Oct 16 12:00:17 1996
 Copyright (c) 1993–1996    Texas Instruments Incorporated

module2.abs                                                          PAGE    1

      15 2003                 .text
      16                      .copy       ”module2.asm”
 A     1 8066               .bss   offset, 2
 A     2                    .copy  globals.def
 B     1                    .global dflag
 B     2                    .global array
 B     3                    .global offset
 A     3 2003 F020          ld     #offset, A
         2004 8066–
 A     4 2005 F020          ld     #array, A
         2006 8000!

 No Errors, No Warnings
```

Chapter 11

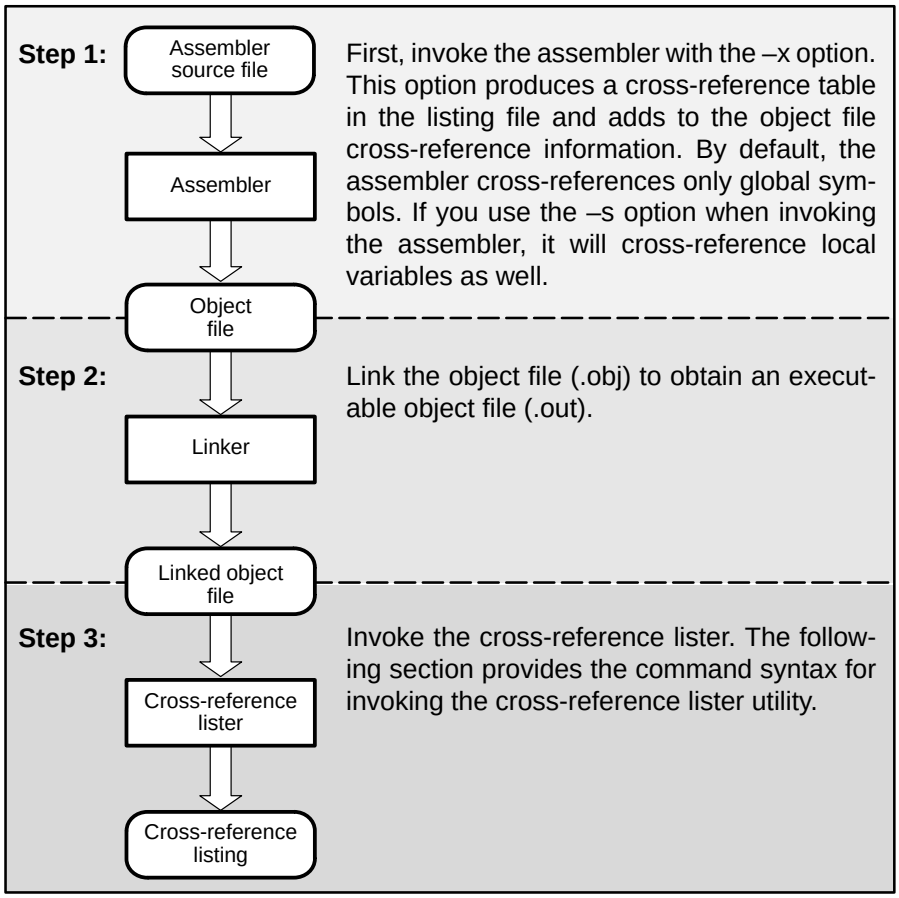
Cross-Reference Lister Description

The cross-reference lister is a debugging tool. This utility accepts linked object files as input and produces a cross-reference listing as output. This listing shows symbols, their definitions, and their references in the linked source files.

Topic **Page**

11.1 Producing a Cross-Reference Listing

Figure 11–1. Cross-Reference Lister Development Flow



11.2 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 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:

xref500 [*–options*] [*input filename* [*output filename*]]

xref500 is the command that invokes the cross-reference utility.

options identifies the cross-reference lister options you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen (–). The cross-reference lister options are as follows:

–l (lowercase L) specifies the number of lines per page for the output file. The format of the –l option is –l*num*, where num is a decimal constant. For example, –l30 sets the number of lines per page in the output file to 30. The space between the option and the decimal constant is optional. The default is 60 lines per page.

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

input filename is a linked object file. If you omit the input filename, the utility prompts for a filename.

output filename is the name of the cross-reference listing file. If you omit the output filename, the default filename will be the input filename with an .xrf extension.

11.3 Cross-Reference Listing Example

```
================================================================================

Symbol: done

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
x.asm            STAT    '000c       380c        18      14

================================================================================

Symbol: f1

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
x.asm            STAT   3.45e+00   3.45e+00      4

================================================================================

Symbol: g3

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
x.asm            EDEF     ffff       ffff        3       9
y.asm            EREF     0000       ffff                3        6

================================================================================

Symbol: start

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
x.asm            STAT    '0000       3800        12      17

================================================================================

Symbol: table

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
x.asm            STAT    -1000       3000        21      13

================================================================================

Symbol: y

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
x.asm            EREF     0000       380d                7        10
y.asm            EDEF    '0000       380d          5     1

================================================================================
```

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

Symbol Name 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:

- **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 a global.
- **UNDF** The symbol is not defined in this file and is not declared as global.

AsmVal This hexadecimal number is the value assigned to the symbol at assembly time. A value may also be preceded by a character that describes the symbol's attributes. Table 11–1 lists these characters and names.

LnkVal This hexadecimal number is the 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. If the line number is followed by a letter (such as A, B, or C), the symbol is referenced in a file specified by a .include directive in the assembly source. "A" is assigned to the first file specified by a .include directive; "B" is assigned to the second file, etc. A blank in this column indicates that the symbol was never used.

Table 11–1. 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

Chapter 12

Hex Conversion Utility Description

The TMS320C54x 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). This utility also supports the on-chip boot loader built into the target device, automating the code creation process for the ’C54x.

The hex conversion utility can produce these output file formats:

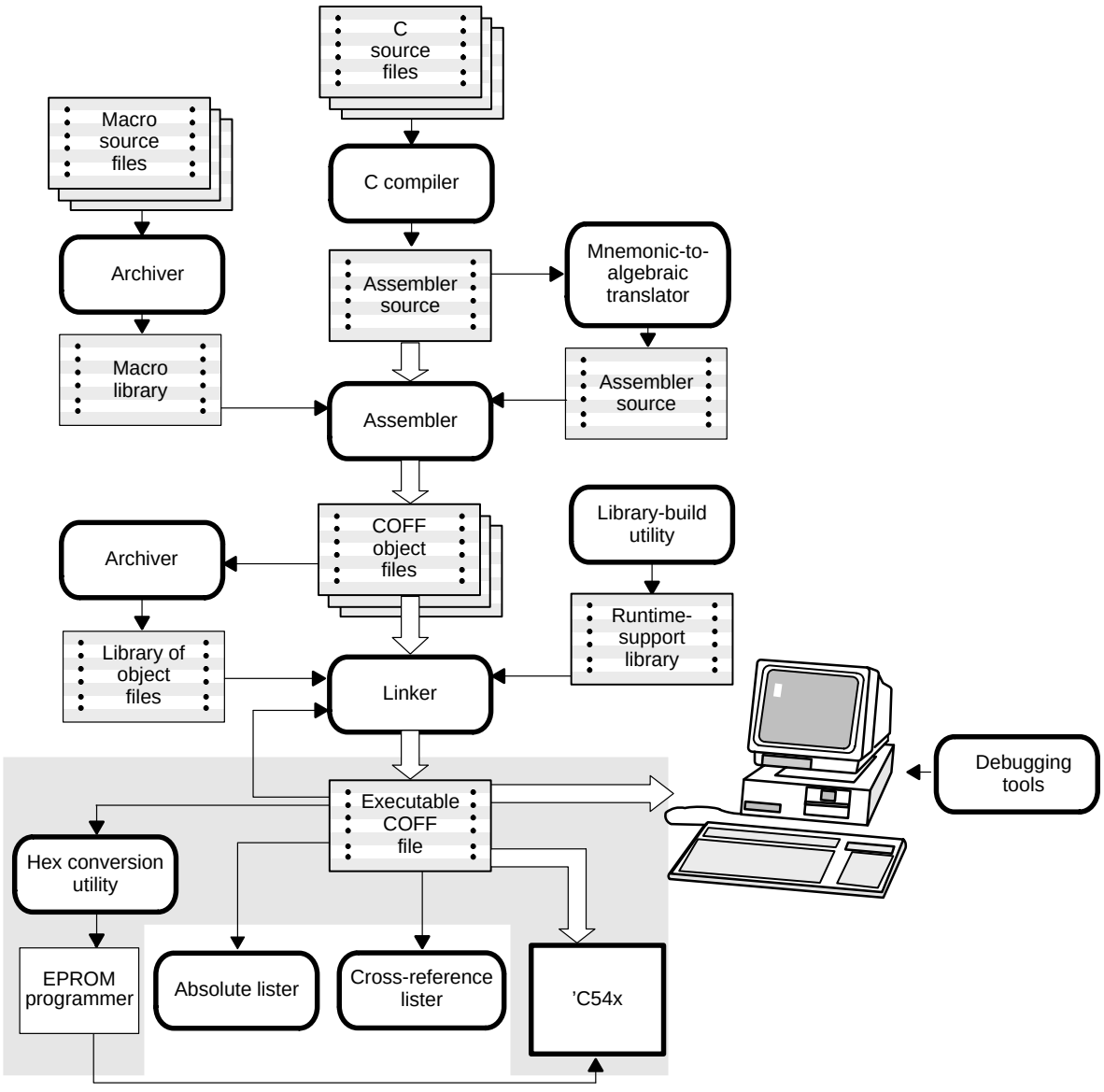
- ASCII-Hex, supporting 16-bit addresses
- Extended Tektronix (Tektronix)
- Intel MCS-86 (Intel)
- Motorola Exorciser (Motorola-S), supporting 16-bit, 24–bit, and 32–bit addresses
- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses

Topic **Page**

12.1 Hex Conversion Utility Development Flow

Figure 12–1 highlights the role of the hex conversion utility in the assembly language development process.

Figure 12–1. Hex Conversion Utility Development Flow



12.2 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.

  ```
  hex500 –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:

  ```
  hex500 hexutil.cmd
  ```

  In addition to regular command line information, you can use the hex conversion utility ROMS and SECTIONS directives in a command file.

To invoke the hex conversion utility, enter:

hex500 [*–options*] *filename*

hex500 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.

- 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.

Table 12–1 lists all the options and directs you to detailed information. Table 12–2 on page 12-29 lists options that apply only to the on-chip boot loader. The boot loader is discussed in Section 12.9.3, *Building a Table for an On-Chip Boot Loader*, on page 12-29.

filename names a COFF object file or a command file (for more information on command files, see Section 12.3, *Command Files*, on page 12-7).

Table 12–1. Options

(a) General options

The general options control the overall operation of the hex conversion utility.

Option	Description	Page
–byte	Number bytes sequentially	12-36
–map *filename*	Generate a map file	12-21
–o *filename*	Specify an output filename	12-24
–q	Run quietly (when used, it must appear *before* other options)	12-7

(b) Image options

The image options create a continuous image of a range of target memory.

Option	Description	Page
–fill *value*	Fill holes with *value*	12-27
–image	Specify image mode	12-26
–zero	Reset the address origin to zero	12-35

(c) Memory options

The memory options configure the memory widths for your output files.

Option	Description	Page
–memwidth *value*	Define the system memory word width (default 16 bits)	12-10
–order {LS | MS}	Specify the memory word ordering	12-14
–romwidth *value*	Specify the ROM device width (default depends on format used)	12-11

(d) Output formats

The output formats specify the format of the output file.

Option	Description	Page
–a	Select ASCII-Hex	12-39
–i	Select Intel	12-40
–m1	Select Motorola–S1	12-41
–m2 or –m	Select Motorola–S2 (default)	12-41
–m3	Select Motorola–S3	12-41
–t	Select TI-Tagged	12-42
–x	Select Tektronix	12-43

(e) Boot-loader options for all 'C54x devices

The boot-loader options for all 'C54x devices control how the hex conversion utility builds the boot table.

Option	Description	Page
–boot	Convert all sections into bootable form (use instead of a SECTIONS directive)	12-29
–bootorg PARALLEL	Specify the source of the boot loader table as the parallel port	12-29
–bootorg SERIAL	Specify the source of the boot loader table as the serial port	12-29
–bootorg *value*	Specify the source address of the boot loader table	12-29
–bootpage *value*	Specify the target page number of the boot loader table	12-29
–e *value*	Specify the entry point for the boot loader table	12-29

(f) Boot-loader options for the 'C54xLP devices only

The boot-loader options for 'C54xLP devices control how the hex conversion utility builds the boot table.

Option	Description	Page
–bootorg WARM	Specify the source of the boot loader table as the table currently in memory	12-29
–bootorg COMM	Specify the source of the boot loader table as the communications port	12-29
–spc *value*	Set the serial port control register value	12-29
–spce *value*	Set the serial port control extension register value	12-29
–arr *value*	Set the ABU receive address register value	12-29
–bkr *value*	Set the ABU transmit buffer size register value	12-29
–tcsr *value*	Set the TDM serial port channel select register value	12-29
–trta *value*	Set the TDM serial port receive/transmit address register value	12-29
–swwsr *value*	Set the Software Wait State Reg value for PARALLEL/WARM boot mode	12-29
–bscr *value*	Set the Bank-Switch Control Reg value for PARALLEL/WARM boot mode	12-29

12.3 Command File

A command file is useful if you plan to invoke the utility more than once with the same input files and options. It is also useful if you want to use the ROMS and SECTIONS hex conversion utility directives to customize the conversion process.

Command files are ASCII files that contain one or more of the following:

- **Options and filenames.** These are specified in a command file in exactly the same manner as on the command line.
- **ROMS directive.** The ROMS directive defines the physical memory configuration of your system as a list of address-range parameters. (For more information about the ROMS directive, see Section 12.5, *The ROMS Directive*, on page 12-16.)
- **SECTIONS directive.** The SECTIONS directive specifies which sections from the COFF object file should be selected. (For more information about the SECTIONS directive, see Section 12.6, *The SECTIONS Directive*, on page 12-22.) You can also use this directive to identify specific sections that will be initialized by an on-chip boot loader. (For more information on the on-chip boot loader, see Section 12.9.3, *Building a Table for an On-Chip Boot Loader*, on page 12-29.)
- **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:

hex500 *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:

```
hex500 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.

12.3.1 Examples of Command Files

- 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:

```
hex500 firmware.cmd
```

- This example converts a file called appl.out into four hex files in Intel format. Each output file is one byte wide and 16K bytes long. The .text section is converted to boot loader format.

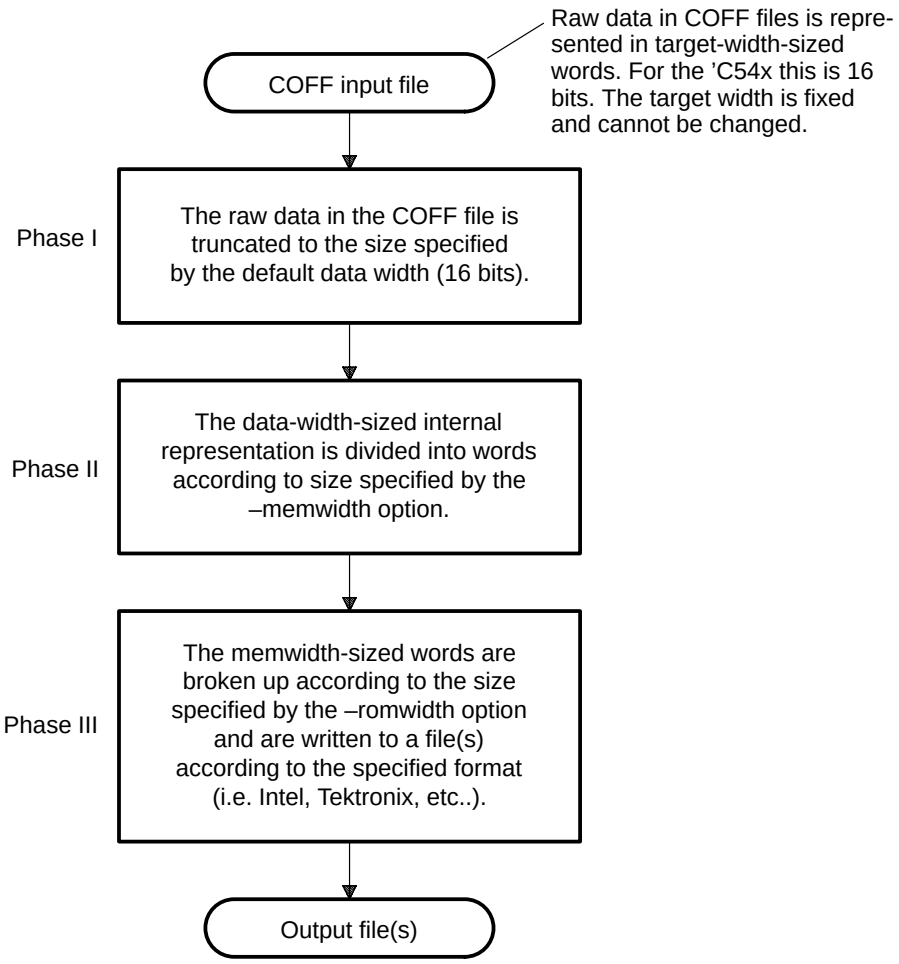
```
appl.out         /* input file   */
-i               /* Intel format */
-map appl.mxp    /* map file     */

ROMS
{
   ROW1: origin=01000h len=04000h romwidth=8
         files={ appl.u0 appl.u1 }
   ROW2: origin 05000h len=04000h romwidth=8
         files={ app1.u2 appl.u3 }
}

SECTIONS
{   .text: BOOT
    .data, .cinit, .sect1, .vectors, .const:
}
```

12.4 Understanding Memory Widths

The hex conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. In order to use the hex conversion utility, *you must understand how the utility treats word widths*. Four widths are important in the conversion process: target width, data width, memory width, and ROM width. The terms target word, data word, memory word and ROM word, refer to a word of such a width.

Figure 12–2 illustrates the three separate and distinct phases of the hex conversion utility's process flow.

Figure 12–2. Hex Conversion Utility Process Flow



12.4.1 Target Width

Target width is the unit size (in bits) of raw data fields in the COFF file. This corresponds to the size of an opcode on the target processor. The width is fixed for each target and cannot be changed. The TMS320C54x targets have a width of 16 bits.

12.4.2 Data Width

Data width is the logical width (in bits) of the data words stored in a particular section of a COFF file. Usually, the logical data width is the same as the target width. The data width is fixed at 16 bits for the TMS320C54x and cannot be changed.

12.4.3 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 16-bit processor has a 16-bit memory architecture. However, some applications, such as boot loaders, require target words to be broken up into multiple, consecutive, narrower memory words. Moreover, with certain processors like the ’C54x, the memory width can be narrower than the target width.

The hex conversion utility defaults memory width to the target width (in this case, 16 bits).

You can change the memory width by:

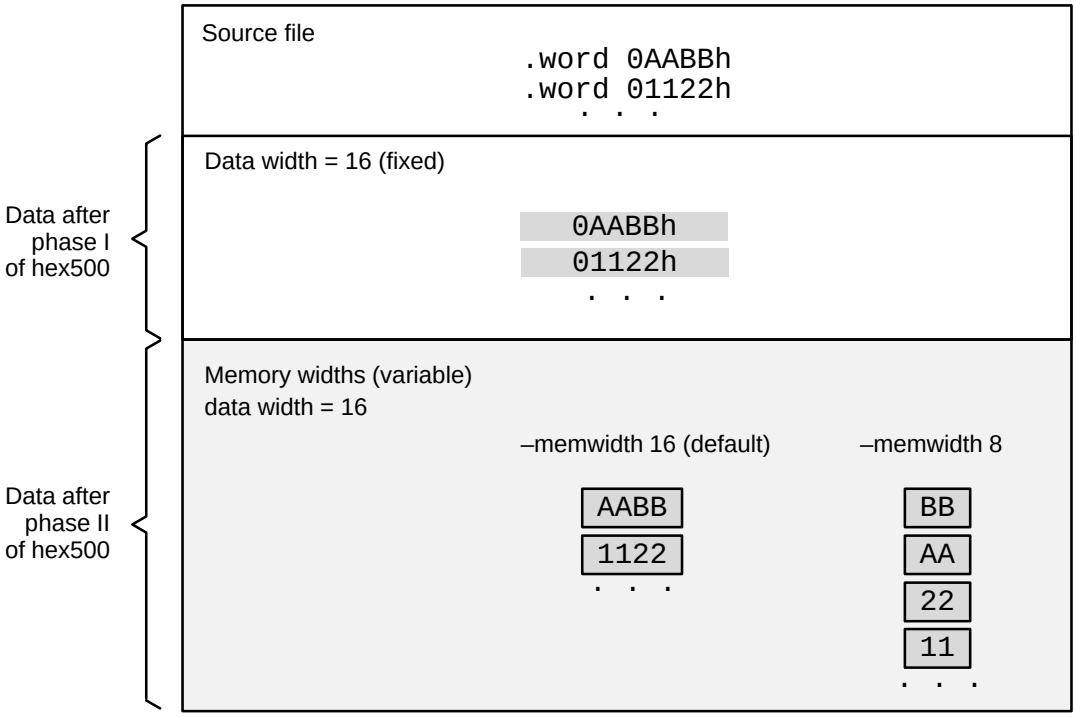
- Using the **–memwidth** option. This changes the memory width value for the entire file.
- Setting the **memwidth** parameter of the ROMS directive. This changes the memory width value for the address range specified in the ROMS directive and overrides the –memwidth option for that range. See Section 12.5, *The ROMS Directive*, on page 12-16.

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 16 only in exceptional situations: for example, when you need to break single target words into consecutive, narrower memory words. Situations in which memory words are narrower than target words are most common when you use on-chip boot loaders—several of which support booting from narrower memory. For example, a 16-bit TMS320C54x can be booted from 8-bit memory or an 8-bit serial port, with each 16-bit value occupying two memory locations (this would be specified as –memwidth 8).

Figure 12–3 demonstrates how the memory width is related to the data width.

Figure 12–3. Data and Memory Widths



12.4.4 ROM Width

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 target words are 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 formula, where memory width ≥ ROM width:

number of files = memory width ÷ ROM width

For example, for a memory width of 16, you could specify a ROM width of 16 and get a single output file containing 16-bit words. Or you can use a ROM width value of 8 to get two files, each containing 8 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) by:

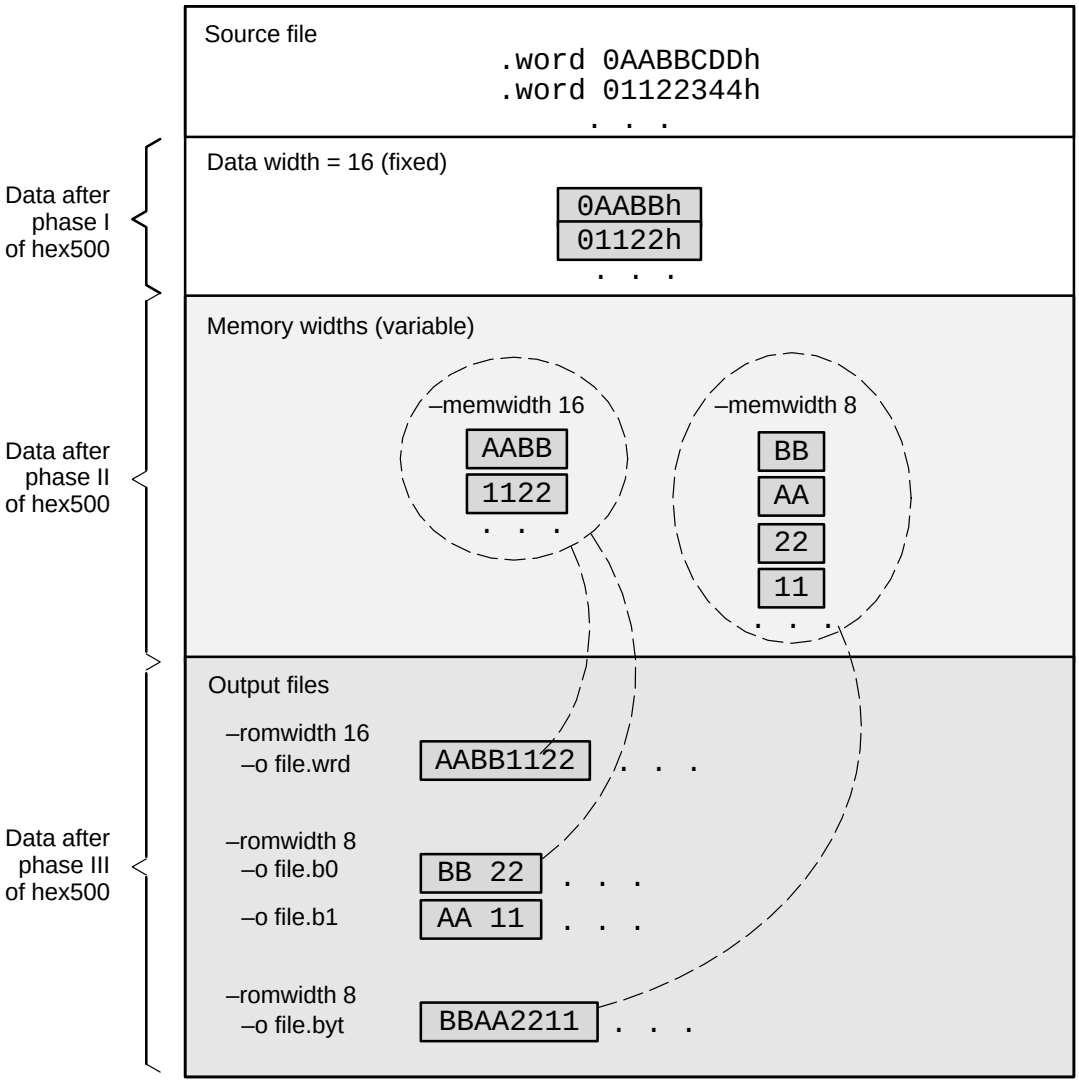
- Using the **–romwidth** option. This changes the ROM width value for the entire COFF file.
- Setting the **romwidth** parameter of the ROMS directive. This changes the ROM width value for a specific ROM address range and overrides the –romwidth option for that range. See Section 12.5, *The ROMS Directive*, on page 12-16.

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 12–4 illustrates how the target, memory, and ROM widths are related to one another.

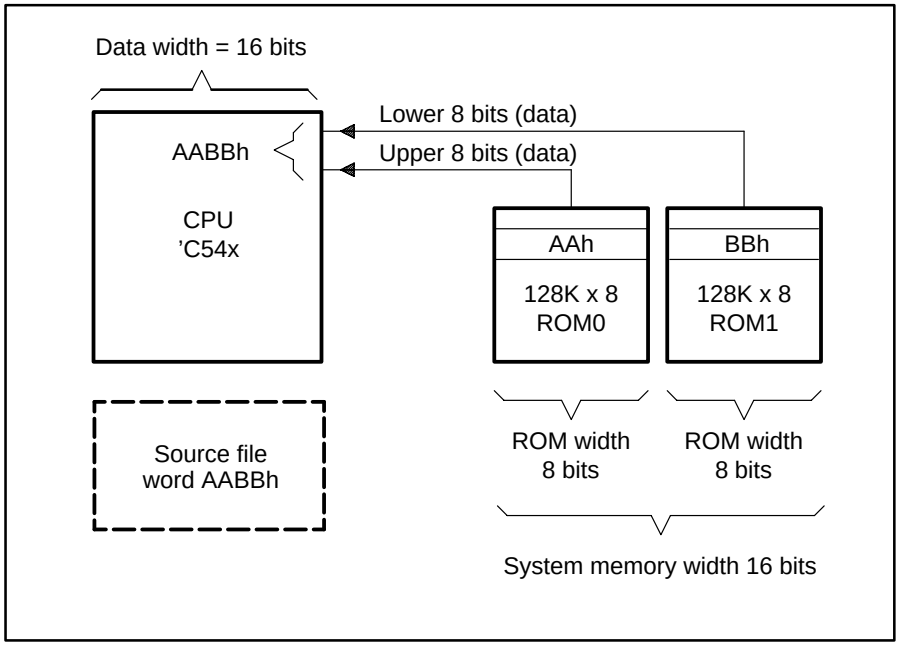
Figure 12–4. Data, Memory, and ROM Widths



12.4.5 A Memory Configuration Example

Figure 12–5 shows a typical memory configuration example. This memory system consists of two 128K × 8-bit ROM devices.

Figure 12–5. 'C54x Memory Configuration Example



12.4.6 Specifying Word Order for Output Words

When memory words are narrower than target words (memory width < 16), target words are split into multiple consecutive memory words. There are two ways to split a wide word into consecutive memory locations in the same hex conversion utility output file:

- **–order MS** specifies **big-endian** ordering, in which the most significant part of the wide word occupies the first of the consecutive locations
- **–order LS** specifies **little-endian** ordering, in which the the least significant part of the wide word occupies the first of the consecutive locations

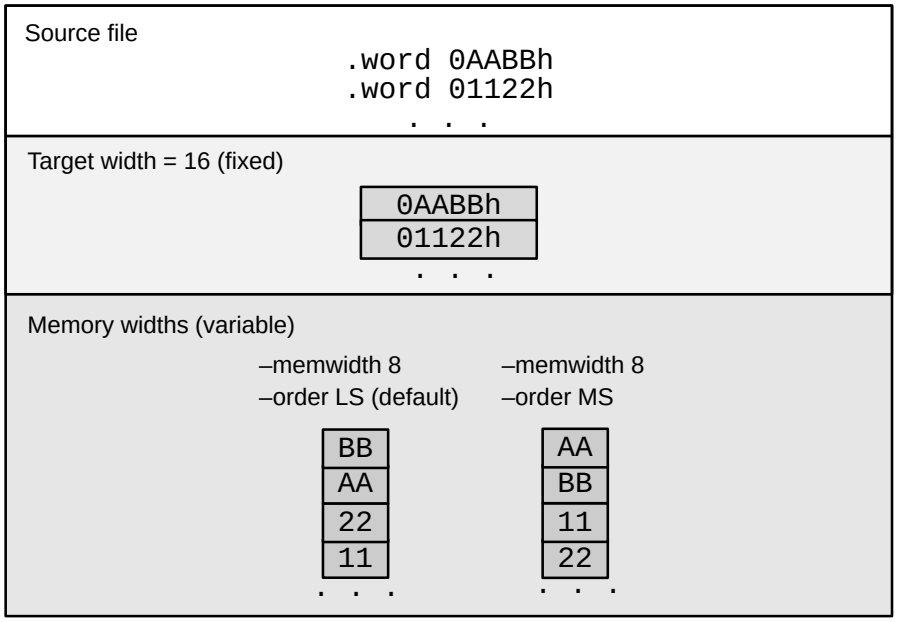
By default, the utility uses little-endian format because the 'C54x boot loaders expect the data in this order. Unless you are using your own boot loader program, avoid the using –order MS.

Note: When the –order Option Applies

- This option applies only when you use a memory width with a value less than 16. Otherwise, –order is ignored.
- This option does not affect the way memory words are split into output files. Think of the files as a set: the set contains a least significant file and a most significant file, but there is no ordering over the set. When you list filenames for a set of files, you *always* list the least significant first, regardless of the –order option.

Figure 12–6 demonstrates how –order affects the conversion process. This figure, and the previous figure, Figure 12–4, explain the condition of the data in the hex conversion utility output files.

Figure 12–6. Varying the Word Order



12.5 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 TMS320C54x 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
{
   [PAGE n:]
      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.

PAGE identifies a memory space for targets that use program- and data-address spaces. If your program has been linked normally, PAGE 0 specifies program memory and PAGE 1 specifies data memory. Each memory range after the PAGE command belongs to that page until you specify another PAGE. If you don't include PAGE, all ranges belong to page 0.

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 subsection 12.4.4, *ROM Width*, on page 12-11). Any value you specify here overrides the –romwidth option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8.

memwidth specifies the memory width of the range in bits (see subsection 12.4.3, *Memory Width*, on page 12-10). Any value you specify here overrides the –memwidth option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8. *When using the memwidth parameter, you must also specify the paddr parameter for each section in the SECTIONS directive.*

fill 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. The value must be a decimal, octal, or hexadecimal constant with a width equal to the target width. Any value you specify here overrides the –fill option. When using fill, you must also use the –image command line option. See subsection 12.8.2, *Specifying a Fill Value*, on page 12-27.

files identifies the names of the output files that correspond to this range. Enclose the list of names in curly braces and order them from *least significant* to *most significant* output file.

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 12.4.4, *ROM Width*, on page 12-11. The utility warns you if you list too many or too few filenames.

Unless you are using the –image option, all of the parameters defining a range are optional; the commas and equals 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 on the same page must not overlap and must be listed in order of ascending address.

12.5.1 When to Use the ROMS Directive

If you don't use a ROMS directive, the utility defines a single default range that includes the entire program address space (PAGE 0). 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. Gaps 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.

12.5.2 An Example of the ROMS Directive

The ROMS directive in Example 12–1 shows how 16K words of 16-bit memory could be partitioned for four 8K × 8-bit EPROMs.

Example 12–1. A ROMS Directive Example

```
infile.out
–image
–memwidth 16

ROMS
{
   EPROM1: org = 04000h, len = 02000h, romwidth = 8
           files = { rom4000.b0, rom4000.b1 }

   EPROM2: org = 06000h, len = 02000h, romwidth = 8,
           fill = 0FFh,
           files = { rom6000.b0, rom6000.b1 }
}
```

In this example, EPROM1 defines the address range from 4000h through 5FFFh. The range contains the following sections:

This section	Has this range
.text	4000h through 487Fh
.data	5B80H through 5FFFh

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 6000h through 7FFFh. The range contains the following sections:

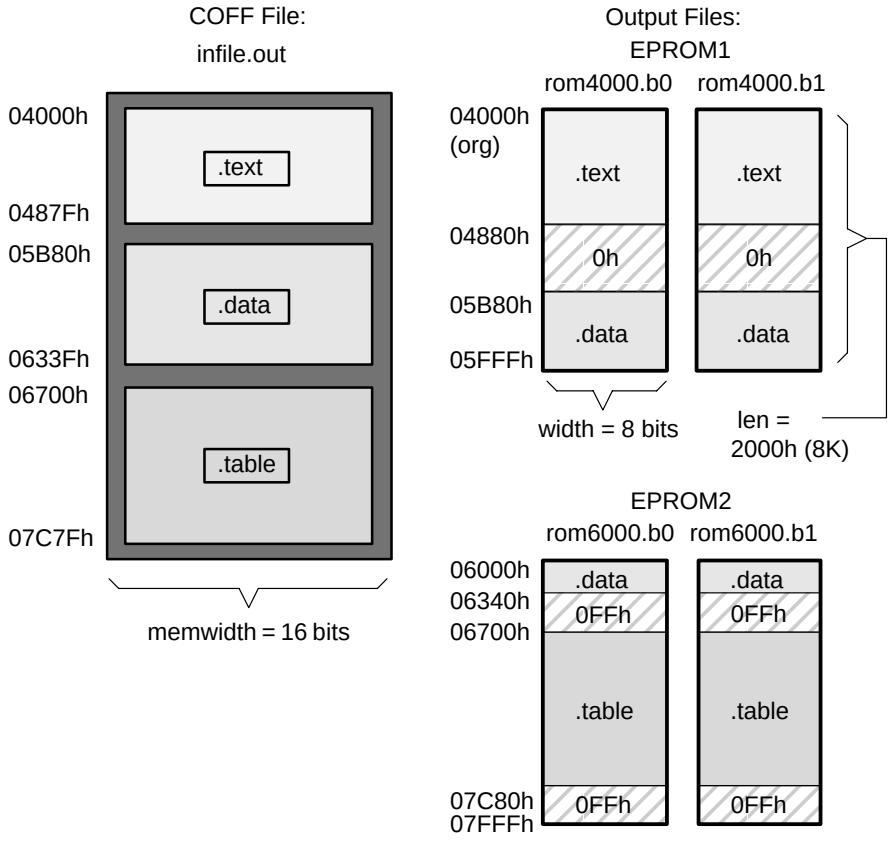
This section	Has this range
.data	6000h through 633Fh
.table	6700h through 7C7Fh

The rest of the range is filled with 0FFh (from the specified fill value). The data from this range is converted into two output files:

- rom6000.b0 contains bits 0 through 7
- rom6000.b1 contains bits 8 through 15

Figure 12–7 shows how the ROMS directive partitions the infile.out file into four output files.

Figure 12–7. The infile.out File From Example 12–1 Partitioned Into Four Output Files



12.5.3 Creating a Map File of the ROMS Directive

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. Following is a segment of the map file resulting from the example in Example 12–1.

Example 12–2. Map File Output From Example 12–1 Showing Memory Ranges

```
--------------------------------------------------
00004000..00005fff  Page=0  Width=8  ”EPROM1”
--------------------------------------------------
   OUTPUT FILES:   rom4000.b0   [b0..b7]
                   rom4000.b1   [b8..b15]

   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
--------------------------------------------------
00006000..00007fff  Page=0  Width=8  ”EPROM2”
--------------------------------------------------
   OUTPUT FILES:   rom6000.b0   [b0..b7]
                   rom6000.b1   [b8..b15]

   CONTENTS: 00006000..0000633f .data
             00006340..000066ff FILL = 000000ff
             00006700..00007c7f .table
             00007c80..00007fff FILL = 000000ff
```

12.6 The SECTIONS Directive

You can convert specific sections of the COFF file by name with the SECTIONS directive. You can also specify those sections you want the utility to configure for loading from an on-chip boot loader, and 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.
- Don't use a SECTIONS directive, the utility converts all initialized sections that fall within the configured memory. The TMS320C54x compiler-generated initialized sections include: .text, .const, .cinit, and .switch.

Uninitialized sections are *never* converted, whether or not you specify them in a SECTIONS directive.

Note: Sections Generated by the C Compiler

The TMS320C54x C compiler automatically generates these sections:

- **Initialized sections:** .text, .const, .cinit, and .switch.
- **Uninitialized sections:** .bss, .stack, and .sysmem.

Use the SECTIONS directive in a command file. (For more information about using a command file, see Section 12.3, *Command Files*, on page 12-7.) The general syntax for the SECTIONS directive is:

```
SECTIONS
{
   sname: [paddr=value]
   sname: [paddr=boot]
   sname: [= boot ],
   ...
}
```

SECTIONS begins the directive definition.

sname identifies a section in the COFF input file. If you specify a section that doesn't exist, the utility issues a warning and ignores the name.

paddr specifies the physical ROM address at which this section should be located. This value overrides the section load address given by the linker. (See Section 12.10, *Controlling the ROM Device Address*, on page 12-34).This value must be a decimal, octal, or hexadecimal constant; it can also be the word **boot** (to indicate a boot table section for use with the on-chip boot loader). *If your file contains multiple sections, and if one section uses a paddr parameter, then all sections must use a paddr parameter.*

= boot configures a section for loading by the on-chip boot loader. This is equivalent to using **paddr=boot**. Boot sections have a physical address determined both by the target processor type and by the various boot-loader-specific command line options.

The commas are optional. For similarity with the linker's SECTIONS directive, you can use colons after the section names (in place of the equal sign on the boot keyboard). For example, the following statements are equivalent:

```
SECTIONS { .text: .data: boot }
SECTIONS { .text, .data = boot }
```

In another 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 }
```

To configure both of these sections for boot loading, add the boot keyword:

```
SECTIONS { .text = boot, .data = boot }
```

Note: Using the –boot Option and the SECTIONS Directive

When you use the SECTIONS directive with the on-chip boot loader, the –boot option is ignored. You must explicitly specify any boot sections in the SECTIONS directive. For more information about –boot and other command line options associated with the on-chip boot loader, see Table 12–2, page 12-29.

12.7 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 data into byte-wide or word-wide files, *filenames are always assigned in order from least to most significant.* This is true, regardless of target or COFF endian ordering, or of any –order option.

12.7.1 Assigning Output Filenames

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 16-bit words being converted to two 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 }
   }
   ```

   The utility creates the output files by writing the least significant bits (LSBs) to xyz.b0 and the most significant bits (MSBs) to xyz.b1.

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
   ```

   Note that 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:

      a for ASCII-Hex
      i for Intel
      t for TI-Tagged
      m for Motorola-S
      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 for a 16-bit target processor and you are creating Intel format output. With no output filenames specified, the utility produces two output files named coff.i0 and coff.i1.

   If you include the following ROMS directive when you invoke the hex conversion utility, you would have two output files:

```
ROMS
{
   range1: o = 1000h l = 1000h
   range2: o = 2000h l = 1000h
}
```

These Output Files	Contain This Data
coff.i01	1000h through 1FFFh
coff.i11	2000h through 2FFFh

12.8 Image Mode and the –fill Option

This section points out the advantages of operating in image mode and describes how to produce output files with a precise, continuous image of a target memory range.

12.8.1 The –image Option

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 gaps 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 gaps by using the address records in the output file to skip ahead to the start of the next section. In other words, there may be discontinuities in the output file addresses. Some EPROM programmers do not support address discontinuities.

In image mode, there are no discontinuities. Each output file contains a continuous stream of data that corresponds exactly to an address range in target memory. Any gaps 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 will always be 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 don't 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.

12.8.2 Specifying a Fill Value

The –fill option specifies a value for filling the holes between sections. The fill value must be specified as an integer constant following the –fill option. The width of the constant is assumed to be that of a word on the target processor. For example, for the ’C54x, specifying –fill 0FFh results in a fill pattern of 00FFh. The constant value is not sign extended.

The hex conversion utility uses a default fill value of zero if you don’t specify a value with the fill option. *The –fill option is valid only when you use –image*; otherwise, it is ignored.

12.8.3 Steps to Follow in Image Mode

Step 1: Define the ranges of target memory with a ROMS directive. See Section 12.5, *The ROMS Directive*, on page 12-16 for details.

Step 2: Invoke the hex conversion utility with the –image option. To number the bytes sequentially, use the –byte option; to reset the address origin to zero for each output file, use the –zero option. See subsection 12.10.3, *The –byte Option*, on page 12-36 for details on the –byte option, and page 12-35 for details on the –zero option. If you don’t specify a fill value with the ROMS directive and you want a value other than the default of zero, use the –fill option.

12.9 Building a Table for an On-Chip Boot Loader

Some DSP devices, such as the ’C54x, have a built-in boot loader that initializes memory with one or more blocks of code or data. The boot loader uses a special table (a *boot table*) stored in memory (such as EPROM) or loaded from a device peripheral (such as a serial or communications port) to initialize the code or data. The hex conversion utility supports the boot loader by automatically building the boot table.

12.9.1 Description of the Boot Table

The input for a boot loader is the boot table. The boot table contains records that instruct the on-chip loader to copy blocks of data contained in the table to specified destination addresses. Some boot tables also contain values for initializing various processor control registers. The boot table can be stored in memory or read in through a device peripheral.

The hex conversion utility automatically builds the boot table for the boot loader. Using the utility, you specify the COFF sections you want the boot loader to initialize, the table location, and the values for any control registers. The hex conversion utility identifies the target device type from the COFF file, builds a complete image of the table according to the format required by that device, and converts it into hexadecimal in the output files. Then, you can burn the table into ROM or load it by other means.

The boot loader supports loading from memory that is narrower than the normal width of memory. For example, you can boot a 16-bit TMS320C54x from a single 8-bit EPROM by using the –memwidth option to configure the width of the boot table. The hex conversion utility automatically adjusts the table’s format and length. See the boot loader example in the *TMS320C54x User’s Guide* for an illustration of a boot table.

12.9.2 The Boot Table Format

The boot table format is simple. Typically, there is a header record containing the width of the table and possibly some values for various control registers. Each subsequent block has a header containing the size and destination address of the block followed by data for the block. Multiple blocks can be entered; a termination block follows the last block. Finally, the table can have a footer containing more control register values. See the boot loader section in the *TMS320C54x User’s Guide* for more information.

12.9.3 How to Build the Boot Table

Table 12–2 summarizes the hex conversion utility options available for the boot loader.

Table 12–2. Boot-Loader Options

(a) Options for all 'C54x devices

Option	Description
–boot	Convert all sections into bootable form (use instead of a SECTIONS directive)
–bootorg PARALLEL	Specify the source of the boot loader table as the parallel port
–bootorg SERIAL	Specify the source of the boot loader table as the serial port
–bootorg *value*	Specify the source address of the boot loader table
–bootpage *value*	Specify the target page number of the boot loader table
–e *value*	Specify the entry point for the boot loader table

(b) Options for 'C54xLP devices only

Option	Description
–bootorg WARM	Specify the source of the boot loader table as the table currently in memory
–bootorg COMM	Specify the source of the boot loader table as the communications port
–spc *value*	Set the serial port control register value
–spce *value*	Set the serial port control extension register value
–arr *value*	Set the ABU receive address register value
–bkr *value*	Set the ABU transmit buffer size register value
–tcsr *value*	Set the TDM serial port channel select register value
–trta *value*	Set the TDM serial port receive/transmit address register value
–swwsr *value*	Set the software wait state register value for PARALLEL/ WARM boot mode
–bscr *value*	Set the bank-switch control register value for PARALLEL/ WARM boot mode

12.9.3.1 Building the Boot Table

To build the boot table, follow these steps:

Step 1: **Link the file**. Each block of the boot table data corresponds to an initialized section in the COFF file. Uninitialized sections are not converted by the hex conversion utility (see Section 12.6, *The SECTIONS Directive*, on page 12-22).

When you select a section for placement in a boot-loader table, the hex conversion utility places the section's *load address* in the destination address field for the block in the boot table. The section content is then treated as raw data for that block.

The hex conversion utility does not use the section run address. When linking, you need not worry about the ROM address or the construction of the boot table—the hex conversion utility handles this.

Step 2: **Identify the bootable sections**. You can use the –boot option to tell the hex conversion utility to configure all sections for boot loading. Or, you can use a SECTIONS directive to select specific sections to be configured (see Section 12.6, *The SECTIONS Directive*, on page 12-22). Note that if you use a SECTIONS directive, the –boot option is ignored.

Step 3: **Set the ROM address of the boot table**. Use the –bootorg option to set the source address of the complete table. For example, if you are using the 'C54x and booting from memory location 8000h, specify –bootorg 8000h. The address field in the the hex conversion utility output file will then start at 8000h.

If you use –bootorg SERIAL or –bootorg PARALLEL, or if you do not use the –bootorg option at all, the utility places the table at the origin of the first memory range in a ROMS directive. If you do not use a ROMS directive, the table will start at the first section load address. There is also a –bootpage option for starting the table somewhere other than page 0.

Step 4: **Set boot-loader-specific options.** Set such options as entry point and memory control registers as needed.

Step 5: **Describe your system memory configuration**. See Section 12.4, *Understanding Memory Widths*, on page 12-9 and Section 12.5, *The ROMS Directive*, on page 12-16 for details.

12.9.3.2 Leaving Room for the Boot Table

The complete boot table is similar to a single section containing all of the header records and data for the boot loader. The address of this “section” is the boot table origin. As part of the normal conversion process, the hex conversion utility converts the boot table to hexadecimal format and maps it into the output files like any other section.

Be sure to leave room in your system memory for the boot table, especially when you are using the ROMS directive. The boot table cannot overlap other nonboot sections or unconfigured memory. Usually, this is not a problem; typically, a portion of memory in your system is reserved for the boot table. Simply configure this memory as one or more ranges in the ROMS directive, and use the –bootorg option to specify the starting address.

12.9.4 Booting From a Device Peripheral

You can choose to boot from a serial or parallel port by using the SERIAL or PARALLEL keyword with the –bootorg option. Your selection of a keyword depends on the target device and the channel you want to use. For example, to boot a ’C54x from its serial port, specify –bootorg SERIAL on the command line or in a command file. To boot a ’C54x from one of its parallel ports, specify –bootorg PARALLEL.

Note: On-Chip Boot Loader Concerns

- **Possible memory conflicts.** When you boot from a device peripheral, the boot table is not actually in memory; it is being received through the device peripheral. However, as explained in Step 3 on page 12-30, a memory address is assigned.

  If the table conflicts with a nonboot section, put the boot table on a different page. Use the ROMS directive to define a range on an unused page and the –bootpage option to place the boot table on that page. The boot table will then appear to be at location 0 on the dummy page.

- **Why the System Might Require an EPROM Format for a Peripheral Boot Loader Address.** In a typical system, a parent processor boots a child processor through that child’s peripheral. The boot loader table itself may occupy space in the memory map of the parent processor. The EPROM format and ROMS directive address correspond to those used by the parent processor, not those that are used by the child.

12.9.5 Setting the Entry Point for the Boot Table

After completing the boot load process, program execution starts at the address of the first block loaded (the default entry point). By using the –e option with the hex conversion utility, you can set the entry point to a different address.

For example, if you want your program to start running at address 0123h after loading, specify –e 0123h on the command line or in a command file. You can determine the –e address by looking at the map file that the linker generates.

Note: Valid Entry Points

The value must be a constant; the hex conversion utility cannot evaluate symbolic expressions like c_int00 (default entry point assigned by the TMS320C54x C compiler).

When you use the –e option, the utility builds a dummy block of length 1 and data value 0 that loads at the specified address. Your blocks follow this dummy block. Since the dummy block is loaded first, the dummy value of 0 is overwritten by the subsequent blocks. Then, the boot loader jumps to the –e option address after the boot load is completed.

When using the –bootorg WARM option, the –e option sets the address of where the boot table is loaded in ROM.

12.9.6 Using the ’C54x Boot Loader

This subsection explains and gives an example on using the hex conversion utility with the boot loader for ’C54x devices. The ’C54x boot loader has six different modes. You can select these modes by using the –bootorg and –memwidth options:

Mode	–bootorg Setting	–memwidth Setting
8-bit parallel I/O	–bootorg PARALLEL	–memwidth 8
16-bit parallel I/O	–bootorg PARALLEL	–memwidth 16
8-bit serial RS232	–bootorg SERIAL	–memwidth 8
16-bit serial RS232	–bootorg SERIAL	–memwidth 16
8-bit parallel EPROM	–bootorg 0x8000	–memwidth 8
16-bit parallel EPROM	–bootorg 0x8000	–memwidth 16
8-bit parallel	–bootorg WARM	–memwidth 8
16-bit parallel	–bootorg WARM	–memwidth 16
8-bit I/O	–bootorg COMM	–memwidth 8

You should set the –romwidth equal to the –memwidth unless you want to have multiple output files.

The ’C54x can boot through either the serial or parallel interface with either 8- or 16-bit data. The format is the same for any combination: the boot table consists of a field containing the destination address, a field containing the length, and a block containing the data.

You can boot only one section. If you are booting from an 8-bit channel, 16-bit words are stored in the table with the MSBs first; the hex conversion utility automatically builds the table in the correct format.

- To boot from a serial port, specify –bootorg SERIAL when invoking the utility. Use either –memwidth 8 or –memwidth 16.
- To load from a parallel I/O port, invoke the utility by specifying –bootorg PARALLEL. Use either –memwidth 8 or –memwidth 16.
- To boot from external memory (EPROM), specify the source address of the boot memory by using the –bootorg option. Use either –memwidth 8 or –memwidth 16.

For example, the command file in Figure 12–8 allows you to boot the .text section of abc.out from a byte-wide EPROM at location 0x8000.

Figure 12–8. Sample Command File for Booting From a ’C54x EPROM

```
abc.out             /* input file                        */
–o abc.i            /* output file                       */
–i                  /* Intel format                      */
–memwidth 8         /* 8-bit memory                      */
–romwidth 8         /* outfile is bytes, not words       */
–bootorg 0x8000     /* external memory boot              */

SECTIONS { .text: BOOT }
```

12.10 Controlling the ROM Device Address

The hex conversion utility output address field corresponds to the ROM device address. The EPROM programmer burns the data into the location specified by the hex conversion utility output file address field. The hex conversion utility offers some mechanisms to control the starting address in ROM of each section and/or to control the address index used to increment the address field. However, many EPROM programmers offer direct control of the location in ROM in which the data is burned.

12.10.1 Controlling the Starting Address

Depending on whether or not you are using the boot loader, the hex conversion utility output file controlling mechanisms are different.

- **Nonboot-loader mode.** The address field of the hex conversion utility output file is controlled by the following mechanisms listed from low to high priority:

  1) **The linker command file.** By default, the address field of the hex conversion utility output file is a function of the load address (as given in the linker command file) and the hex conversion utility parameter values. The relationship is summarized as follows:

  $$\text{out\_file\_addr}^{\dagger} = \text{load\_addr} \times (\text{data\_width} \div \text{mem\_width})$$

out_file_addr	is the address of the output file.
load_addr	is the linker-assigned load address.
data_width	is specified as 16 bits for the TMS320C54x devices. See subsection 12.4.2, *Data Width*, on page 12-10.
mem_width	is the memory width of the memory system. You can specify the memory width by the –memwidth option or by the memwidth parameter inside the ROMS directive. See subsection 12.4.3, *Memory Width*, on page 12-10.

  † If paddr is not specified

  The value of data width divided by memory width is a correction factor for address generation. When data width is larger than memory width, the correction factor *expands* the address space. For example, if the load address is 0×1 and data width divided by memory width is 2, the output file address field would be 0×2. The data is split into two consecutive locations the size of the memory width.

2) **The paddr parameter of the SECTIONS directive.** When the paddr parameter is specified for a section, the hex conversion utility bypasses the section load address and places the section in the address specified by paddr. The relationship between the hex conversion utility output file address field and the paddr parameter can be summarized as follows:

$$\text{out\_file\_addr}^{\dagger} = \text{paddr\_val} + (\ \text{load\_addr} - \text{sect\_beg\_load\_addr}\ ) \times (\text{data\_width} \div \text{mem\_width})$$

out_file_addr	is the address of the output file.
paddr_val	is the value supplied with the paddr parameter inside the SECTIONS directive.
sec_beg_load_addr	is the section load address assigned by the linker.

† If paddr is not specified

The value of data width divided by memory width is a correction factor for address generation. The section beginning load address factor subtracted from the load address is an offset from the beginning of the section.

3) **The –zero option.** When you use the –zero option, the utility resets the address origin to 0 for each output file. Since each file starts at 0 and counts upward, any address records represent offsets from the beginning of the file (the address within the ROM) rather than actual target addresses of the data.

You must use the –zero option in conjunction with the –image option to force the starting address in each output file to be zero. If you specify the –zero option without the –image option, the utility issues a warning and ignores the –zero option.

- **Boot-Loader Mode.** When the boot loader is used, the hex conversion utility places the different COFF sections that are in the boot table into consecutive memory locations. Each COFF section becomes a boot table block whose destination address is equal to the linker-assigned section load address.

  The address field of the the hex conversion utility output file is not related to the section load addresses assigned by the linker. The address fields are simply offsets to the beginning of the table, multiplied by the correction factor (data width divided by memory width).

The beginning of the boot table defaults to the linked load address of the first bootable section in the COFF input file, unless you use one of the following mechanisms, listed here from low to high priority. Higher priority mechanisms override the values set by low priority options in an overlapping range.

1) **The ROM origin specified in the ROMS directive.** The hex conversion utility places the boot table at the origin of the first memory range in a ROMS directive.

2) **The –bootorg option.** The hex conversion utility places the boot table at the address specified by the –bootorg option if you select boot loading from memory. Neither –bootorg PARALLEL nor –bootorg SERIAL affect the address field.

12.10.2 Controlling the Address Increment Index

By default, the hex conversion utility increments the output file address field according to the memory width value. If memory width equals 16, the address increments on the basis of how many 16-bit words are present in each line of the output file.

12.10.3 The –byte Option

Some EPROM programmers may require the output file address field to contain a byte count rather than a word count. If you use the –byte option, the output file address increments once for each byte. For example, if the starting address is 0h, the first line contains eight words, and you use no –byte option, the second line would start at address 8 (8h). If the starting address is 0h, the first line contains eight words, and you use the –byte option, the second line would start at address 16 (010h). The data in both examples are the same; –byte affects only the calculation of the output file address field, not the actual target processor address of the converted data.

The –byte option causes the address records in an output file to refer to byte locations within the file, whether the target processor is byte-addressable or not.

12.10.4 Dealing With Address Holes

When memory width is different from data width, the automatic multiplication of the load address by the correction factor might create holes at the beginning of a section or between sections.

For example, assume you want to load a COFF section (.sec1) at address 0x0100 of an 8-bit EPROM. If you specify the load address in the linker command file at location 0x0100, the hex conversion utility will multiply the address by 2 (data width divided by memory width = 16/8 = 2), giving the output file a starting address of 0x0200. Unless you control the starting address of the EPROM with your EPROM programmer, you could create holes within the EPROM. The programmer will burn the data starting at location 0x0200 instead of 0x0100. To solve this, you can:

- **Use the paddr parameter of the SECTIONS directive.** This forces a section to start at the specified value. Figure 12–9 shows a command file that can be used to avoid the hole at the beginning of .sec1.

Figure 12–9. Hex Command File for Avoiding a Hole at the Beginning of a Section

```
–i
a.out
–map a.map

ROMS
{
 ROM : org = 0x0100, length = 0x200, romwidth = 8,
       memwidth = 8
}

SECTIONS
 {
   sec1: paddr = 0x100
}
```

Note: If your file contains multiple sections, and, if one section uses a paddr parameter, then all sections must use the paddr parameter.

- **Use the –bootorg option or use the ROMS origin parameter (for boot loading only).** As described on page 12-35, when you are boot loading, the EPROM address of the entire boot-loader table can be controlled by the –bootorg option or by the ROMS directive origin. For another example, see Section C.4, *Example 3: Generating a Boot Table for Non-LP Core Devices*, on page C-10.

12.11 Description of the Object Formats

The hex conversion utility converts a COFF object file into one of five object formats that most EPROM programmers accept as input: ASCII-Hex, Intel MCS-86, Motorola-S, Extended Tektronix, or TI-Tagged.

Table 12–3 specifies the format options.

- If you use more than one of these options, the last one you list overrides the others.
- The default format is Tektronix (–x option).

Table 12–3. Options for Specifying Hex Conversion Formats

Option	Format	Address Bits	Default Width
–a	ASCII-Hex	16	8
–i	Intel	32	8
–m1	Motorola-S1	16	8
–m2 or –m	Motorola-S2	24	8
–m3	Motorola-S3	32	8
–t	TI-Tagged	16	16
–x	Tektronix	32	8

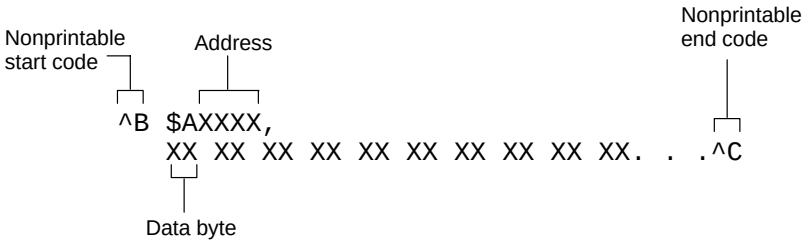
Address bits determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

The **default width** determines the default output width. 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.

12.11.1 ASCII-Hex Object Format (–a Option)

The ASCII-Hex object format supports 16-bit addresses. The format consists of a byte stream with bytes separated by spaces. Figure 12–10 illustrates the ASCII-Hex format.

Figure 12–10. 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. The output created is a list of byte values.

12.11.2 Intel MCS-86 Object Format (–i Option)

The Intel object format supports 16-bit addresses and 32-bit extended addresses. Intel format consists of a 9-character (4-field) prefix—which defines the start of record, byte count, load address, and record type—the data, and a 2-character checksum suffix.

The 9-character prefix represents three record types:

Record Type	Description
00	Data record
01	End-of-file record
04	Extended linear address record

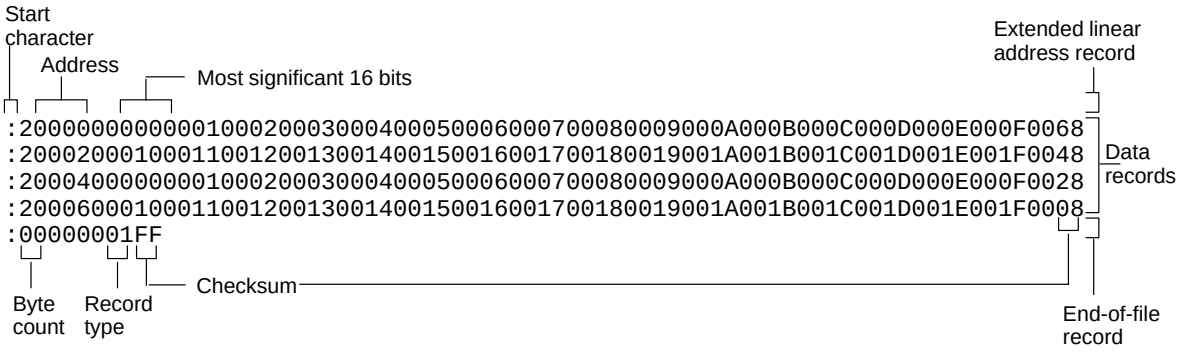
Record type *00*, the data record, begins with a colon (:) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. Note that the address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most 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 12–11 illustrates the Intel hexadecimal object format.

Figure 12–11. Intel Hex Object Format



12.11.3 Motorola Exorciser Object Format (–m1, –m2, –m3 Options)

The Motorola S1, S2, and S3 formats support 16-bit, 24–bit, and 32–bit addresses, respectively. The formats consist of a start-of-file (header) record, data records, and an end-of-file (termination) record. Each record is made up of five fields: record type, byte count, address, data, and checksum. The record types are:

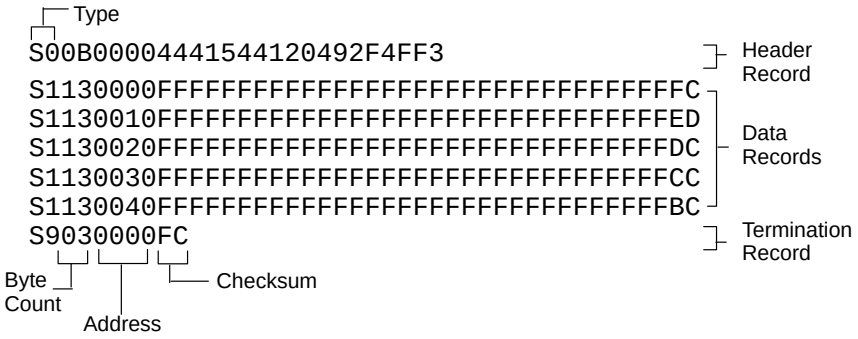
Record Type	Description
S0	Header record
S1	Code/data record for 16–bit addresses (S1 format)
S2	Code/data record for 24–bit addresses (S2 format)
S3	Code/data record for 32–bit addresses (S3 format)
S7	Termination record for 32–bit addresses (S3 format)
S8	Termination record for 24–bit addresses (S2 format)
S9	Termination record for 16–bit addresses (S1 format)

The byte count is the character pair count in the record, excluding the type and byte count itself.

The checksum is the least significant byte of the 1s complement of the sum of the values represented by the pairs of characters making up the byte count, address, and the code/data fields.

Figure 12–12 illustrates the Motorola-S object format.

Figure 12–12. Motorola-S Format



12.11.4 Texas Instruments SDSMAC Object Format (–t Option)

The TI-Tagged object format supports 16-bit addresses. It consists of a start-of-file record, data records, and end-of-file record. Each of the data records is made up of a series of small fields and is signified by a tag character. The significant tag characters are:

Tag Character	Description
K	followed by the program identifier
7	followed by a checksum
8	followed by a dummy checksum (ignored)
9	followed by a 16-bit load address
B	followed by a data word (four characters)
F	identifies the end of a data record
*	followed by a data byte (two characters)

Figure 12–13 illustrates the tag characters and fields in TI-Tagged object format.

Figure 12–13. TI-Tagged Object Format

If any data fields appear before the first address, the first field is assigned address 0000h. Address fields may be expressed for any data byte, but none is required. The checksum field, which is preceded by the tag character 7, is a 2s complement of the sum of the 8-bit ASCII values of characters, beginning with the first tag character and ending with the checksum tag character (7 or 8). The end-of-file record is a colon (:).

12.11.5 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.

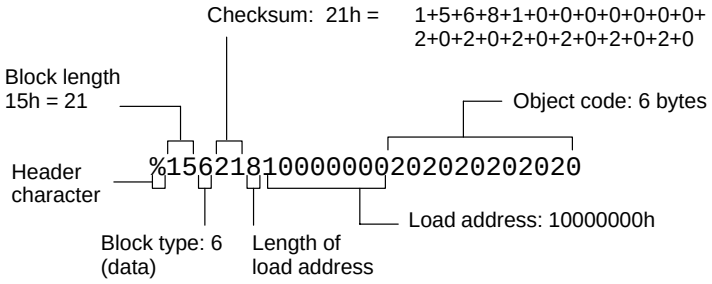
The header field in the data record contains the following information:

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 12–14 illustrates the Tektronix object format.

Figure 12–14. Extended Tektronix Object Format



12.12 Hex Conversion Utility Error Messages

section mapped to reserved memory message

Description A section or a boot-loader table is mapped into a reserved memory area listed in the processor memory map.

Action Correct the section or boot-loader address. Refer to the *TMS320C54x User's Guide* for valid memory locations.

sections overlapping

Description Two or more COFF section load addresses overlap or a boot table address overlaps another section.

Action This problem may be caused by an incorrect translation from load address to hex output file address that is performed by the hex conversion utility when memory width is less than data width. See Section 12.4, *Understanding Memory Widths*, on page 12-9 and Section 12.10, *Controlling the ROM Device Address*, on page 12-34.

unconfigured memory error

Description This error could have one of two causes:

- The COFF file contains a section whose load address falls outside the memory range defined in the ROMS directive.
- The boot-loader table address is not within the memory range defined by the ROMS directive.

Action Correct the ROM range as defined by the ROMS directive to cover the memory range as needed, or modify the section load address or boot-loader table address. Remember that if the ROMS directive is not used, the memory range defaults to the entire processor address space. For this reason, removing the ROMS directive could also be a workaround.

Chapter 13

Mnemonic-to-Algebraic Translator Description

The TMS320C54x mnemonic-to-algebraic translator utility converts assembly code written in the mnemonic instruction set to code written in the algebraic instruction set.

Topic **Page**

13.1 Translator Overview

The TMS320C54x mnemonic-to-algebraic translator utility converts mnemonic assembly instructions into algebraic assembly instructions. Mnemonic instructions usually consist of a keyword and operands. Algebraic instructions usually consist of operands and operators. Algebraic instructions resemble higher-level programming language instructions.

The translator requires error-free code. When the translator encounters unrecognized instructions or macro invocations, it prints a message to standard output and does not translate the line of code.

The translator accepts assembly code source files containing mnemonic instructions and produces assembly code source files containing algebraic instructions. The input file can have no extension or an extension of *asm*. The output file will have the same name as the input file with an extension of *cnv*.

13.1.1 What the Translator Does

The translator accomplishes the following:

- Replaces a mnemonic with an algebraic representation of what the instruction does as defined by the language specifications. The algebraic representation might consist of more than one line of code.
- Reformats mnemonic instruction operands into algebraic syntax as described in the language specifications. This reformatting includes the following:
  - Data memory address (dma) accesses are prefixed with a @ symbol.
  - The mnemonic indirect shorthand * is replaced with *AR0.
  - When necessary, constants are prefixed with a # symbol.
  - Algebraic expressions that are used as a single operand and have more than one term are enclosed in parentheses.

13.1.2 What the Translator Does Not Do

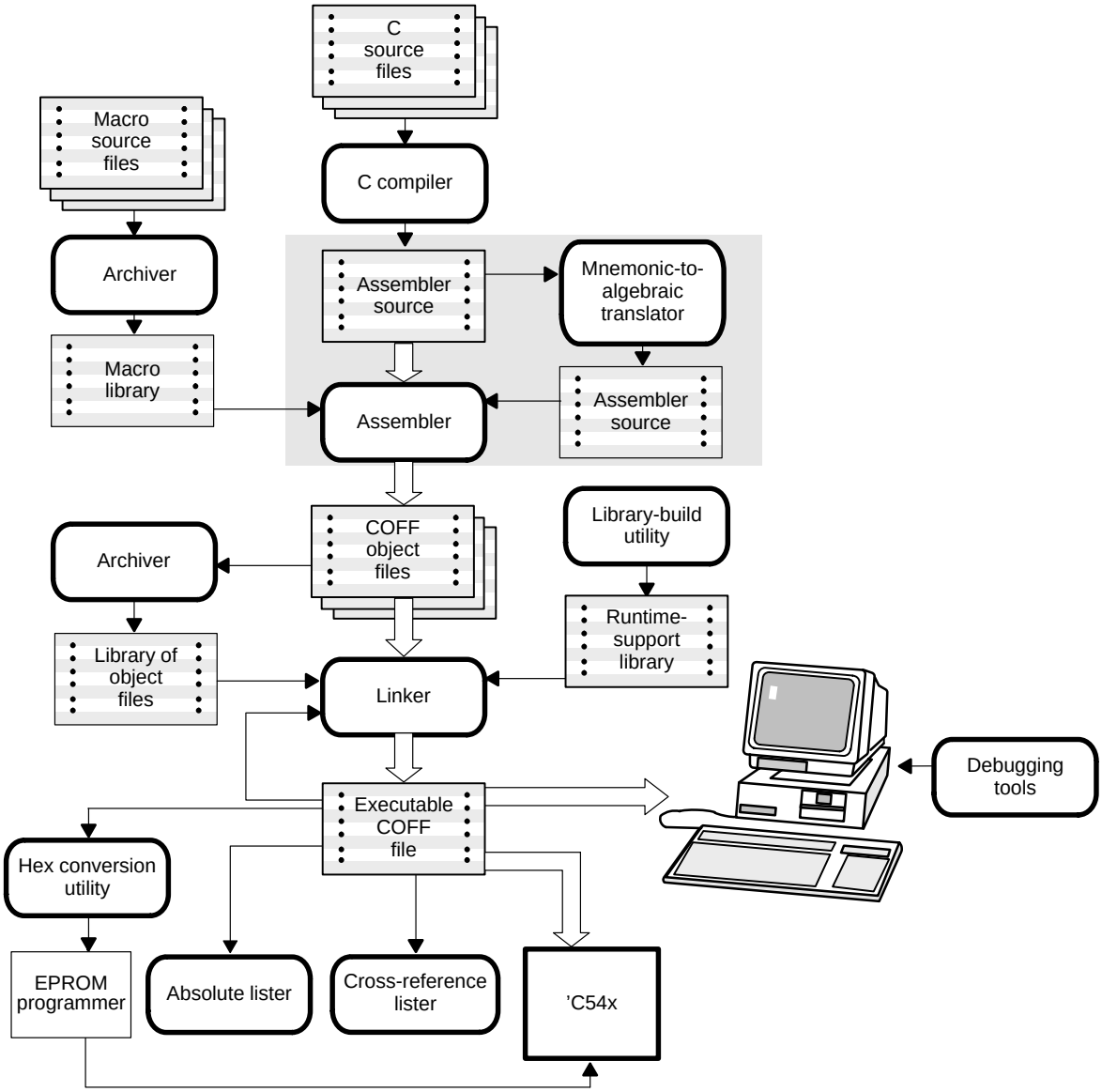
The translator has the following limitations:

- The translator cannot convert macro definitions. It ignores them. Optionally, the translator replaces macro invocations with the expanded macro, replacing the formal parameters with the actual arguments used at invocation.
- The translator attempts to translate any macro that has the same name as a mnemonic instruction. Insure that macro names are different from mnemonic instructions.

13.2 Translator Development Flow

Figure 8–1 shows the translator's role in the assembly language development process. The assembler accepts mnemonic or algebraic syntax.

Figure 13–1. Translator Development Flow



13.3 Invoking the Translator

To invoke the translator, enter:

mnem2alg [*option*] *inputfile*

mnem2alg is the command that invokes the translator.

option specifies the translator mode (see Section 13.4, *Translation Modes*, on page 13-5). The options are:

- **–t** Literal mode, which is the default if no option is specified
- **–e** Expansion mode

inputfile names the assembly source file you want to translate. If you do not specify an extension, *asm* is assumed.

13.4 Translation Modes

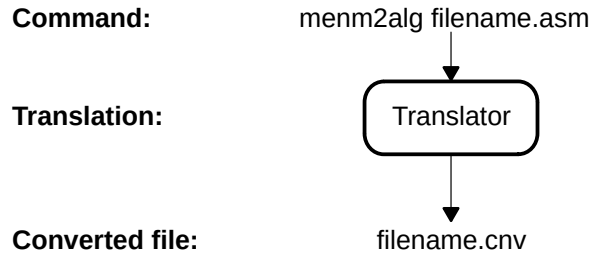
The translator runs in one of the following modes:

Literal	Keeps the original mnemonic instruction, commented out, followed by the translated instruction
Expansion	Expands and preprocesses macro invocations and replaces substitution symbols

13.4.1 Literal Mode (–t Option)

When running in the default literal mode (–t option), the translator translates instructions without any preprocessing. The translator does not process macros, nor does it expand substitution symbols. When the translator does not recognize a macro invocation or instruction, it prints a message to standard output and does not translate the code. The translator creates a file with the same name as the assembler source file and an extension of *cnv*.

Figure 13–2. Literal Mode Process



13.4.2 About Symbol Names in Literal Mode

In literal mode, the translator treats symbol names defined by .asg as labels and not as the value they represent. In the following example, the source code is translated as shown, with sym treated as a data memory address:

Example 13–1. Treatment of Symbol Names in Literal Mode

(a) Source code:

```
sym    .asg   *AR2
       LD  sym,B
```

(b) Converted code:

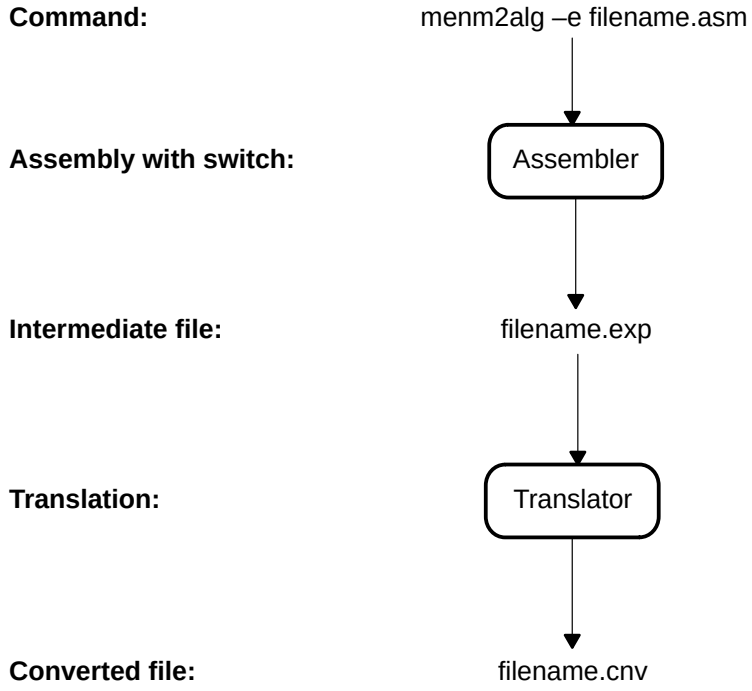
```
sym    .asg *AR2
       B = @sym
```

13.4.3 Expansion Mode (–e Option)

Expansion mode is invoked using the –e option. In expansion mode, the translator preprocesses macros and substitution symbols and then translates instructions. The translator invokes the assembler with a switch that preprocesses the input to expand macros and insert substitution symbols. The assembler creates a file with an *exp* extension. The *exp* file is passed back to the translator for processing. The translator creates a file with the same name as the assembler source file and an extension of *cnv*.

Since the translator invokes the assembler in expansion mode, you must include the assembler executable in your path. The assembler executable version must be 1.11 or above. If the assembler encounters errors during preprocessing, the translator aborts and no output is produced.

Figure 13–3. Expansion Mode Process



The following example demonstrates how expansion mode works. In the intermediate file, the macro invocation is commented out and the expanded macro is inserted in its place, with the actual arguments substituted in. Although the macro definition was not translated, the resulting *cnv* file can be assembled by the algebraic assembler to produce output. The assembler does not process macro definitions (for the same reasons the translator does not translate definitions). The *exp* intermediate file that the assembler produces is deleted after the translation is complete.

Example 13–2. Expansion Mode

(a) Source code

```
file.asm
*******************************
.asg *AR0,sym

mymac     .macro parm1,parm2
          LD     parm1,parm2
          ADD    sym,5,parm2,B
          .endm

          mymac  sym,A
*******************************
```

(b) Intermediate code

```
file.exp – after preprocessing
           before translation
*********************************
          .asg *AR0,sym

mymac     .macro parm1,parm2
          LD     parm1,parm2
          ADD    sym,5,parm2,B
          .endm

;           mymac  sym,A
          LD     *AR0,A
          ADD    *AR0,5,A,B
*********************************
```

(c) Converted code

```
file.cnv – after translation
*******************************
sym       .asg *AR0

mymac     .macro parm1,parm2
          LD     parm1,parm2
          ADD    sym,5,parm2,B
          .endm

;           mymac  sym,A
          A = *AR0
          B = A + *AR0 << 5
*******************************
```

13.5 How the Translator Works With Macros

This section describes how the translator works with macros. The following subjects are discussed:

- Directives in macros
- Macro local variables
- Defining labels when invoking a macro

13.5.1 Directives in Macros

When macro invocations are expanded, directives in macro definitions are not copied to the intermediate file. Instead, the macro is inlined, and the code is no longer in a macro environment. The following source code preprocesses to the intermediate code as shown:

Example 13–3. Directives in Macros

(a) Source code

```
mymac  .macro parm1
       .var temp
       .eval parm1, temp
       .word temp
       .endm

       mymac 5
```

(b) Intermediate code

```
mymac  .macro parm1
       .var   temp
       .eval parm1,temp
       .word temp
       .endm

;        mymac 5
       .word 5
```

13.5.2 Macro Local Variables

When macro local variables are encountered, they are changed so that repeated calls to the macro do not generate identical labels. The following source code preprocesses to the intermediate code as shown:

Example 13–4. Macro Local Variables

(a) Source code

```
mymac  .macro  parm1
lab?   .word   parm1
       .endm

       mymac 4
       mymac 40
       mymac 400
```

(b) Intermediate code

```
mymac     .macro  parm1
lab?      .word   parm1
          .endm

;           mymac 4
lab01     .word   4
;           mymac 40
lab02     .word   40
;           mymac 400
lab03     .word   400
```

The local label name is appended with *nn*, where *nn* is the number of the macro invocation. Insure that there are no other labels that could be identical to a generated macro local label.

13.5.3 Defining Labels When Invoking A Macro

If there is a label associated with a macro invocation, that label is not used after expansion and translation. This is because the label is commented out with the macro invocation. The following source code preprocesses to the intermediate code as shown:

Figure 13–4. Defining Labels

(a) Source code:

```
mymac  .macro
       .word F403
       .endm
LABEL  mymac
```

(b) Intermediate code:

```
 mymac  .macro
        .word F403
        .endm
 ;LABEL    mymac
        .word F403
```

LABEL is not defined when the code is assembled. Insure that label definitions do not appear on the same line as the macro invocations. Rewrite the source code in the example above as follows:

Figure 13–5. Rewritten Source Code

```
mymac  .macro
       .word F403
       .endm
LABEL
       mymac
```

Appendix A

Common Object File Format

The compiler, 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 more 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 will be able to use the assembly language tools more efficiently.

This appendix contains technical details about 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 about the internal format of COFF object files.

Topic **Page**

A.1 COFF File Structure

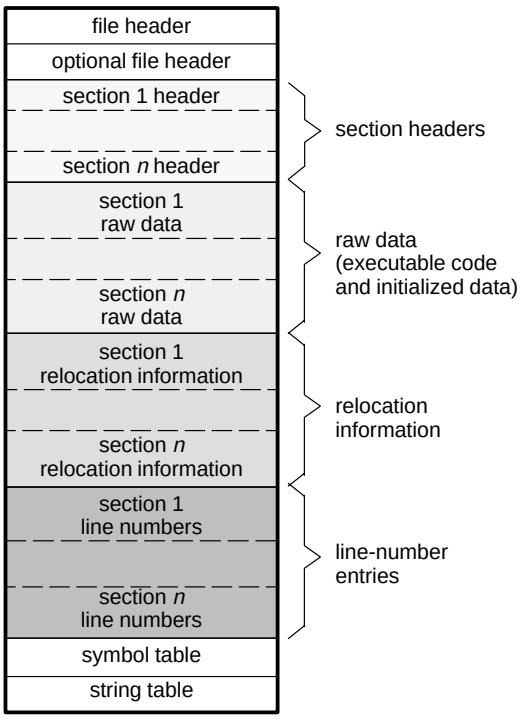
The elements of a COFF object file describe the file's sections and symbolic debugging information. These elements are:

- 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

A.1.1 Overall Object File Structure

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 overall object file structure.

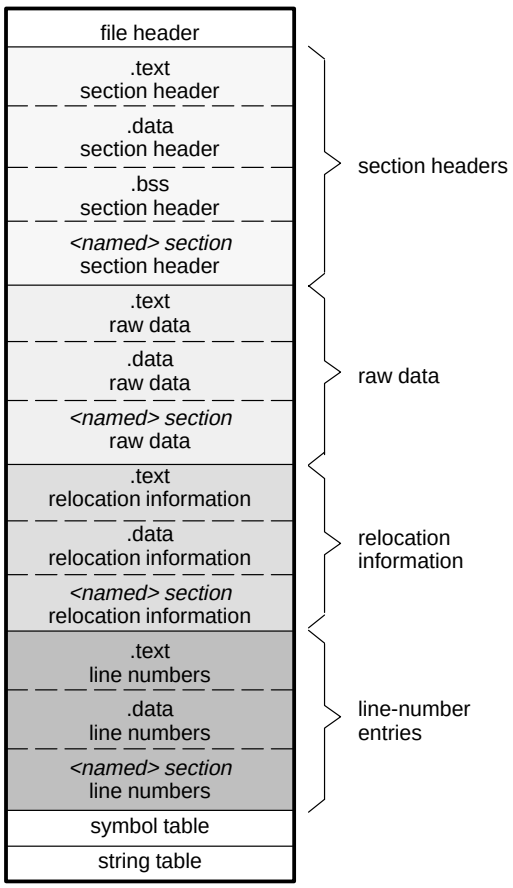
Figure A–1. COFF File Structure



A.1.2 Typical Object 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. COFF Object File



A.1.3 Impact of Switching Operating Systems

The ’C54x COFF files are recognized by all operating system versions of the development tools. When you switch from one operating system to another, only the file header information in the COFF files needs to be byte swapped. The raw data in the COFF files does not need any changes.

The ’C54x development tools can detect the difference in the file headers and automatically compensate for it. This is true if using only ’C54x development tools.

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

A.2 File Header Structure

The file header contains 22 bytes of information that describe the general format of an object file. Table A–1 shows the structure of the 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, then 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 indicates the file can be executed in a TMS320C54x 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	0010h	Local symbols were stripped from the file.
F_LENDIAN	0100h	The file has the byte ordering used by 'C54x devices (16 bits per word, least 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	Magic number (for SunOS or HP-UX it is 108h; for DOS it is 801h)
2–3	Short integer	Version stamp
4–7	Long integer	Size (in words) of executable code
8–11	Long integer	Size (in words) of initialized words
12–15	Long integer	Size (in words) 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. The COFF1 and COFF2 file types contain different section header information. Table A–4 shows the section header contents for COFF1 files. Table A–5 shows the section header contents for COFF2 files.

Table A–4. Section Header Contents for COFF1 Files

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

Table A–5. Section Header Contents for COFF2 Files

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

Table A–6 lists the flags that can appear in the section header. The flags can be combined. For example, if the flag's word is set to 024h, both STYP_GROUP and STYP_TEXT are set.

Table A–6. Section Header Flags

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

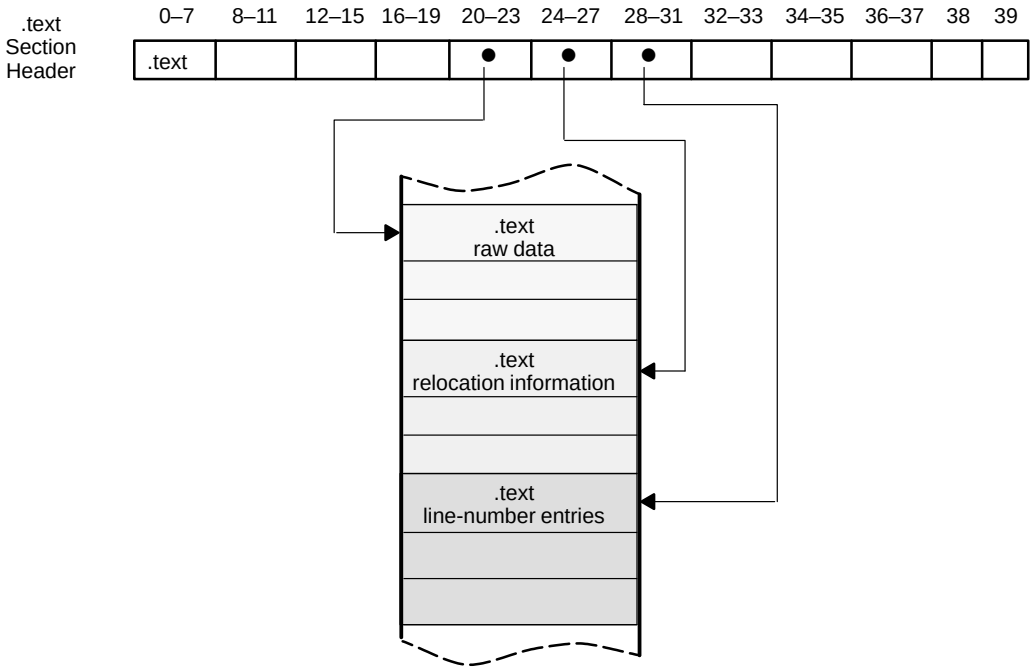
Note: The term *loaded* means that the raw data for this section appears in the object file.

The flags are in:

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

Figure A–3 illustrates how the pointers in a section header would point to the 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 12-byte format shown in Table A–7.

Table A–7. Relocation Entry Contents

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

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

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

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

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

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

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

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

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

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

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_RELBYTE	000Fh	8-bit direct reference to symbol's address
R_REL13	002Ah	13-bit direct reference
R_RELWORD	0010h	16-bit direct reference to symbol's address
R_PARTLS7	0028h	7 LSBs of an address
R_PARTMS9	0029h	9 MSBs of an address

A.6 Line-Number Table Structure

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

Table A–9. Line-Number Entry Format

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

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

Figure A–4. Line-Number Blocks

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

As Figure A–4 shows, each entry is divided as follows:

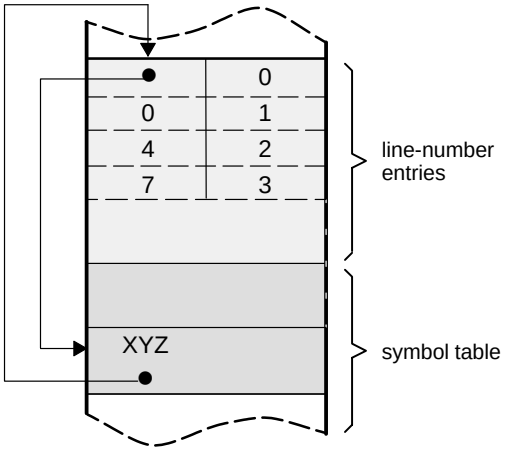
- For the *first line* of a function, bytes 0–3 point to the name of a symbol or a function in the symbol table, and bytes 4–5 contain a 0, which indicates the beginning of a block.

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

The line-number entry table can contain many of these blocks.

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

Figure A–5. Line-Number Entries



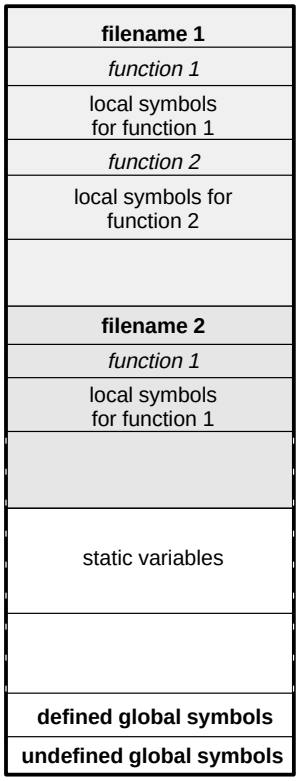
(Note that the symbol table entry for XYZ has a field that points back to the beginning of the line-number block.)

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

A.7 Symbol Table Structure and Content

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

Figure A–6. Symbol Table Contents



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

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

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

Section names are also defined in the symbol table.

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

Table A–10. Symbol Table Entry Contents

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

A.7.1 Special Symbols

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

Table A–11. Special Symbols in the Symbol Table

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

Several of these symbols appear in pairs:

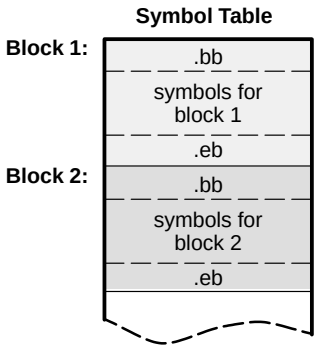
- .bb/.eb indicate the beginning and end of a block.
- .bf/.ef indicate the beginning and end of a function.
- *n*fake/.eos name and define the limits of structures, unions, and enumerations that were not named. The .eos symbol is also paired with named structures, unions, and enumerations.

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

A.7.1.1 Symbols and Blocks

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

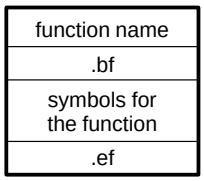
Figure A–7. Symbols for Blocks



A.7.1.2 Symbols and Functions

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

Figure A–8. Symbols for Functions



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

A.7.2 Symbol Name Format

The first 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 8 characters, this field is treated as two long integers. The entire symbol name is stored in the string table. Bytes 0–3 contain 0, and bytes 4–7 are an offset into the string table.

A.7.3 String Table Structure

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

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

Figure A–9. String Table

38			
'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.7.4 Storage Classes

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

Table A–12. Symbol Storage Classes

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

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

Table A–13. Special Symbols and Their Storage Classes

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

A.7.5 Symbol Values

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

Table A–14. Symbol Values and Storage Classes

Storage Class	Value Description	Storage Class	Value Description
C_AUTO	Stack offset in bits	C_UNTAG	0
C_EXT	Relocatable address	C_TPDEF	0
C_STAT	Relocatable address	C_ENTAG	0
C_REG	Register number	C_MOE	Enumeration value
C_LABEL	Relocatable address	C_REGPARM	Register number
C_MOS	Offset in bits	C_FIELD	Bit displacement
C_ARG	Stack offset in bits	C_BLOCK	Relocatable address
C_STRTAG	0	C_FCN	Relocatable address
C_MOU	Offset in bits	C_FILE	0

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

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

A.7.6 Section Number

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

Table A–15. Section Numbers

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

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

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

A.7.7 Type Entry

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

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

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

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

Table A–16. Basic Types

Mnemonic	Value	Type
T_NULL	0	Type not assigned
T_CHAR	2	Character
T_SHORT	3	Short integer
T_INT	4	Integer
T_LONG	5	Long integer
T_FLOAT	6	Floating point
T_DOUBLE	7	Double word
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_MOE	11	Member of an enumeration
T_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short integer

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

Table A–17. Derived Types

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

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

A.7.8 Auxiliary Entries

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

Table A–18. Auxiliary Symbol Table Entries Format

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

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

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

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

A.7.8.1 Filenames

Each of the auxiliary table entries for a filename contains a 14-character filename in bytes 0–13. Bytes 14–17 are unused.

Table A–19. Filename Format for Auxiliary Table Entries

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

A.7.8.2 Sections

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

Table A–20. Section Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Section length
4–6	Unsigned short integer	Number of relocation entries
7–8	Unsigned short integer	Number of line-number entries
9–17	—	Not used (zero filled)

A.7.8.3 Tag Names

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

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

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

A.7.8.4 End of Structure

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

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

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

A.7.8.5 Functions

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

Table A–23. Function Format for Auxiliary Table Entries

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

A.7.8.6 Arrays

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

Table A–24. Array Format for Auxiliary Table Entries

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

A.7.8.7 End of Blocks and Functions

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

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

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

A.7.8.8 Beginning of Blocks and Functions

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

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

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

A.7.8.9 Names Related to Structures, Unions, and Enumerations

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

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

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	—	Unused (zero filled)
6–7	Unsigned short integer	Size of the structure, union, or enumeration
8–17	—	Unused (zero filled)
16–17	—	Unused (zero filled)

Appendix B

Symbolic Debugging Directives

The TMS320C54x assembler supports several directives that the TMS320C54x C compiler uses for symbolic debugging:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the symbol 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 function.
- The **.block** and **.endblock** directives specify the bounds of C blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source file name.
- The **.line** directive identifies the line number of a C source statement.

These symbolic debugging directives are not usually listed in the assembly language file that the compiler creates. If you want them to be listed, invoke the compiler shell with the –g option, as shown below:

```
cl500 –g input file
```

This appendix contains an alphabetical directory of the symbolic debugging directives. With the exception of the .file directive, each directive contains an example of C source and the resulting assembly language code.

Syntax

.block *beginning line number*
.endblock *ending line number*

Description

The **.block** and **.endblock** directives specify the beginning and end of a C block. The line numbers are optional; they specify the location in the source file where the block is defined.

Block definitions can be nested. The assembler will detect improper block nesting.

Example

Following is an example of C source that defines a block, and the resulting assembly language code.

C source:

```
  .
  .
  .
{                   /*  Beginning of a block */
     int  a,b;
     a = b;
}                   /*  End of a block       */
  .
  .
  .
```

Resulting assembly language code:

```
.block 6
.sym  _a,0,4,1,16
.sym  _b,1,4,1,16
.line 5
    LD        *SP(1),A              ; cycle 3
    STL       A,*SP(0)              ; cycle 4
.endblock    8
```

Syntax

.file "*filename***"**

Description

The **.file** directive allows a debugger to map locations in memory back to lines in a C source file. The *filename* is the name of the file that contains the original C source program. The first 14 characters of the filename are significant.

You can also use the .file directive in assembly code to provide a name in the file and improve program readability.

Example

In the following example, the filename *text.c* contained the C source that produced this directive.

```
.file    "text.c"
```

Syntax

.func *beginning line number*
.endfunc *ending line number*

Description

The **.func** and **.endfunc** directives specify the beginning and end of a C function. The *line numbers* are optional; they specify the location in the source file where the function is defined. Function definitions cannot be nested.

Example

Following is an example of C source that defines a function, and the resulting assembly language code:

C source:

```
power(x, n)  /*  Beginning of a function */
int x,n;
{
   int i, p;
   p = 1;
   for (i = 1; i <= n; ++i)
      p = p * x;
   return p;        /*  End of function  */
}
```

Resulting assembly language code:

```
 8        .global _power
 9        .sym    _power,_power,36,2,0
10        .func   3
11
12        ;***************************************************************
13        ;* FUNCTION DEF: _power                                        *
14        ;***************************************************************
15 0000       _power:
16 0000 eefd          FRAME     #-3               ; cycle 1
17 0001 f495          nop                     ;* A     assigned to _x
19                    .sym    _x,8,4,17,16
20                    .sym    _n,4,4,9,16
21                    .sym    _x,0,4,1,16
22                    .sym    _i,1,4,1,16
23                    .sym    _p,2,4,1,16
24                    .line   2
25 0002 8000          STL       A,*SP(0)          ; cycle 3
26                    .line   4
27 0003 7602          ST        #1,*SP(2)         ; cycle 4
   0004 0001
28                    .line   5
29 0005 7601          ST        #1,*SP(1)         ; cycle 6
   0006 0001
30 0007 f7b8          SSBX      SXM               ; cycle 8
31 0008 f495          nop
32 0009 1004          LD        *SP(4),A          ; cycle 10
33 000a 0801          SUB       *SP(1),A          ; cycle 11
34 000b f843          BC        L3,ALT            ; cycle 12
   000c 0018'
35                    ; branch occurs             ; cycle 17
36 000d       L2:
37                    .line   6
38 000d 4400          LD        *SP(0),16,A       ; cycle 1
39 000e 3102          MPYA      *SP(2)            ; cycle 2
40 000f 8102          STL       B,*SP(2)          ; cycle 3
41                    .line   5
42 0010 6b01          ADDM      #1,*SP(1)         ; cycle 4
   0011 0001
43 0012 f7b8          SSBX      SXM               ; cycle 6
44 0013 f495          nop
45 0014 1004          LD        *SP(4),A          ; cycle 8
46 0015 0801          SUB       *SP(1),A          ; cycle 9
47 0016 f842          BC        L2,AGEQ           ; cycle 10
   0017 000d'
48                    ; branch occurs             ; cycle 15
49 0018       L3:
50                    .line  7
51 0018 1002          LD        *SP(2),A                ; cycle
 52                     .line   8
53 0019 ee03          FRAME     #3                      ; cycle 1
54 001a fc00          RET                               ; cycle 2
55                    ; branch occurs                   ; cycle 7
56                    .endfunc        10,000000000h,
```

Syntax

.line *line number* [, *address*]

Description

The **.line** directive creates a line number entry in the object file. Line number entries are used in symbolic debugging to associate addresses in the object code with the lines in the source code that generated them.

The .line directive has two operands:

- The *line number* indicates the line of the C source that generated a portion of code. Line numbers are relative to the beginning of the current function. This is a required parameter.
- The *address* is an expression that is the address associated with the line number. This is an optional parameter; if you don't specify an address, the assembler will use the current SPC value.

Example

The .line directive is followed by the assembly language source statements that are generated by the indicated line of C source. For example, assume that the lines of C source below are line 4 and 5 in the original C source; line 5 produces the assembly language source statements that are shown below.

C source:

```
for (i = 1; i <= n; ++i)
     p = p * x;
```

Resulting assembly language code:

```
31                      .line   5
32 000a 4403            LD        *SP(3),16,A           ; cycle 1
33 000b 3101            MPYA      *SP(1)                ; cycle 2
34 000c 8101            STL       B,*SP(1)              ; cycle 3
35                      .line   4
36 000d 6b00            ADDM      #1,*SP(0)             ; cycle 4
   000e 0001
37 000f f7b8            SSBX      SXM                   ; cycle 6
38 0010 f495            nop
39 0011 1002            LD        *SP(2),A              ; cycle 8
40 0012 0800            SUB       *SP(0),A              ; cycle 9
41 0013 f842            BC        L2,AGEQ               ; cycle 10
   0014 000a'
42                      ; branch occurs                 ; cycle 15
43 0015         L3:
44                      .line   7
45 0015 ee05            FRAME     #5                    ; cycle 1
46 0016 fc00            RET                             ; cycle 2
47                      ; branch occurs                 ; cycle 7
48                      .endfunc        9,000000000h,5
49
50
```

Syntax

.member *name, value [, type, storage class, size, tag, dims]*

Description

The **.member** directive defines a member of a structure, union, or enumeration. It is valid only when it appears in a structure, union, or enumeration definition.

- *Name* is the name of the member that is put in the symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the member. Any legal expression (absolute or relocatable) is acceptable.
- *Type* is the C type of the member. Appendix A, *Common Object File Format*, contains more information about C types.
- *Storage class* is the C storage class of the member. Appendix A, *Common Object File Format*, contains more information about C storage classes.
- *Size* is the number of bits of memory required to contain this member.
- *Tag* is the name of the type (if any) or structure of which this member is a type. This name *must* have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be one to four expressions separated by commas. This allows up to four dimensions to be specified for the member.

The order of parameters is significant. The *name* and *value* are required parameters. All other parameters may be omitted or empty. (Adjacent commas indicate an empty entry.) This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example

Following is an example of a C structure definition and the corresponding assembly language statements:

C source:

```
struct doc {
     char  title;
     char  group;
     int   job_number;
}  doc_info;
```

Resulting assembly language code:

```
      .stag   _doc,64
      .member _title,0,2,8,16
      .member _group,16,2,8,16
      .member _job_number,32,4,8,16
      .eos
```

Syntax

```
.stag  name  [, size]
    member definitions
.eos
.etag  name  [, size]
    member definitions
.eos
.utag  name  [, size]
    member definitions
.eos
```

Description

The **.stag** directive begins a structure definition. The **.etag** directive begins an enumeration definition. The **.utag** directive begins a union definition. The **.eos** directive ends a structure, enumeration, or union definition.

- *Name* is the name of the structure, enumeration, or union. The first 32 characters of the name are significant. This is a required parameter.
- *Size* is the number of bits the structure, enumeration, or union occupies in memory. This is an optional parameter; if omitted, the size is unspecified.

The .stag, .etag, or.utag directive should be followed by a number of .member directives, which define members in the structure. The .member directive is the only directive that can appear inside a structure, enumeration, or union definition.

The assembler does not allow nested structures, enumerations, or unions. The C compiler unwinds nested structures by defining them separately and then referencing them from the structure they are referenced in.

Example 1

Following is an example of a structure definition.

C source:

```
struct doc
{
     char  title;
     char  group;
     int   job_number;
}  doc_info;
```

Resulting assembly language code:

```
        .stag    _doc,96
        .member  _title,0,2,8,32
        .member  _group,32,2,8,32
        .member  _job_number,64,4,8,32
        .eos
```

Example 2 Following is an example of a union definition.

C source:

```
union u_tag {
      int   val1;
      float val2;
      char  valc;
}  valu;
```

Resulting assembly language code:

```
            .utag     _u_tag,96
            .member   _val1,0,2,8,32
            .member   _val2,32,4,8,32
            .member   _valc,64,4,8,32
            .eos
```

Example 3 Following is an example of an enumeration definition.

C Source:

```
{
    enum o_ty { reg_1, reg_2, result } optypes;
}
```

Resulting assembly language code:

```
            .etag     _o_ty,32
            .member   _reg_1,0,11,16,32
            .member   _reg_2,1,11,16,32
            .member   _result,2,11,16,32
            .eos
```

Syntax

.sym *name, value [, type, storage class, size, tag, dims]*

Description

The .sym directive specifies symbolic debug information about a global variable, local variable, or a function.

- *Name* is the name of the variable that is put in the object symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the variable. Any legal expression (absolute or relocatable) is acceptable.
- *Type* is the C type of the variable. Appendix A, *Common Object File Format*, contains more information about C types.
- *Storage class* is the C storage class of the variable. Appendix A, *Common Object File Format*, contains more information about C storage classes.
- *Size* is the number of words of memory required to contain this variable.
- *Tag* is the name of the type (if any) or structure of which this variable is a type. This name *must* have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be up to four expressions separated by commas. This allows up to four dimensions to be specified for the variable.

The order of parameters is significant. The *name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example

These lines of C source produce the .sym directives shown below:

C source:

```
struct s { int member1, member2; } str;
int     ext;
int     array[5][10];
long  *ptr;
int     strcmp();

main(arg1,arg2)
   int   arg1;
   char  *arg2;
{
   register r1;
}
```

Resulting assembly language code:

```
.global   _array
.bss      _array,50,0,0
.sym      _array,_array,244,2,800,,5,10
.global   _ptr
.bss      _ptr,1,0,0
.sym      _ptr,_ptr,21,2,16
.global   _str
.bss      _str,2,0,1
.sym      _str,_str,8,2,32,_s
.global   _ext
.bss      _ext,1,0,0
.sym      _ext,_ext,4,2,16
```

Appendix C

Hex Conversion Utility Examples

The flexible hex conversion utility offers many options and capabilities. Once you understand the proper ways to configure the EPROM system and the requirements of the EPROM programmer, you will find that converting a file for a specific application is easy.

Topic **Page**

C.1 Base Code for the Examples

The three major examples in this appendix show how to develop a hex command file for multiple EPROM memory systems, avoid holes, and generate a boot table. The examples use the assembly code shown in Example C–1.

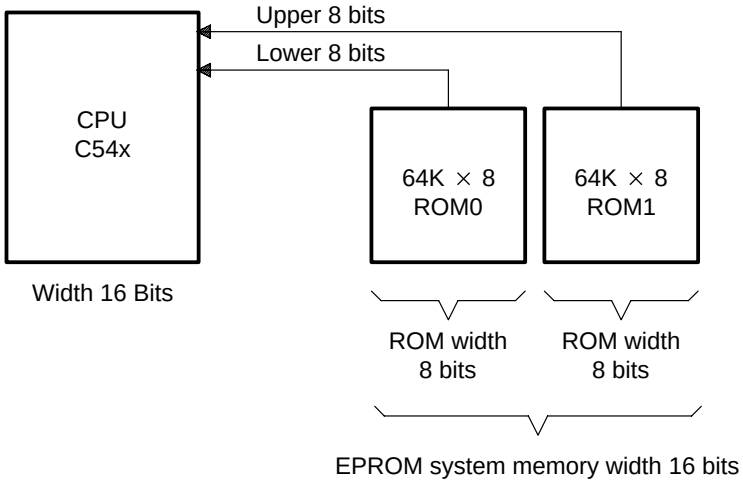
Example C–1. Assembly Code for Hex Conversion Utility Examples

```
*********************************************************
* Assemble two words into section, "sec1"               *
*********************************************************

       .sect "sec1"
       .word 1234h
       .word 5678h

*********************************************************
* Assemble two words into section, "sec2"               *
*********************************************************

       .sect "sec2"
       .word 0aabbh
       .word 0ccddh

       .end
```

C.2 Example 1: Building A Hex Command File for Two 8-Bit EPROMs

Example 1 shows how to build the hex command file you need for converting a COFF object file for the memory system shown in Figure C–1. In this system, there are two external 64K × 8-bit EPROMs interfacing with a ’C54x target processor. Each of the EPROMs contributes 8 bits of a 16-bit word for the target processor.

Figure C–1. A Two 8-Bit EPROM System



By default, the hex conversion utility uses the linker load address as the base for generating addresses in the converted output file. However, for this application, the code will reside at physical EPROM address 0x0010, rather than the address specified by the linker (0x1400). The circuitry of the target board handles the translation of this address space. The paddr parameter allocates a section and burns the code at EPROM address 0x0010.

The paddr parameter is specified within the SECTIONS directive (see Section 12.6, *The SECTIONS Directive*, on page 12-22 for details.). If you use the paddr parameter to specify a load address for one section included in the conversion, then you must specify a paddr for each section included in the conversion. When setting the paddr parameter, you must ensure that the specified addresses do not overlap the linker-assigned load addresses of sections that follow.

In Example 1, two sections are defined: sec1 and sec2. You can easily add a paddr parameter for each of these sections from within the SECTIONS directive. However, the task may become unmanageable for large applications with many sections, or in cases where section sizes may change often during code development.

To work around this problem, you can combine the sections at link stage, creating a single section for conversion. To do this, use the linker command shown in Example C–2.

Example C–2. A Linker Command File for Two 8-Bit EPROMs

```
test.obj
–o test.out
–m test.map

MEMORY
{
   PAGE 0 : EXT_PRG ; org = 0x1400 , len = 0xEB80
}

SECTIONS
{
   outsec: { *(sec1)
             *(sec2)  } > EXT_PRG PAGE 0
}
```

The EPROM programmer in this example has the following system requirements:

- ❑ EPROM system memory width must be 16 bits.
- ❑ ROM1 contains the upper 8 bits of a word.
- ❑ ROM0 contains the lower 8 bits of a word.
- ❑ The hex conversion utility must locate code starting at EPROM address 0x0010.
- ❑ Intel format must be used.
- ❑ Byte increment must be selected for addresses in the hex conversion utility output file (memory width is the default).

Use the following options to set up the requirements of the system:

Option	Description
–i	Create Intel format
–byte	Select byte increment for addresses in converted output file
–memwidth 16	Set EPROM system memory width to 16
–romwidth 8	Set physical ROM width to 8

With the memory width and ROM width values above, the utility will automatically generate two output files. The ratio of memory width to ROM width determines the number of output files. The ROM0 file contains the lower 8 of the 16 bits of raw data, and the ROM1 file contains the upper 8 bits of the corresponding data.

Example C–3 shows the hex command file with all of the selected options.

Example C–3. A Hex Command File for Two 8-Bit EPROMs

```
test.out        /* COFF object input file              */
-map example1.mxp

/*-----------------------------------------------------*/
/* Set parameters for EPROM programmer                 */
/*-----------------------------------------------------*/

-i              /* Select Intel format                 */
-byte           /* Select byte increment for addresses  */

/*-----------------------------------------------------*/
/* Set options required to describe EPROM memory system */
/*-----------------------------------------------------*/

-memwidth 16    /* Set EPROM system memory width       */
-romwidth  8    /* Set physical width of ROM device    */

ROMS
{
   PAGE 0 : EPROM : origin = 0x00, length = 0x10000,
                    files = {low8.bit, upp8.bit}
}

SECTIONS
{ outsec: paddr = 0x10 }
```

Figure C–2 (a) shows the contents of the converted file for ROM0 (low8.bit) containing the lower 8 bits. Figure C–2 (b) shows the contents of the converted file for ROM1 (upp8.bit) containing the upper 8 bits of data.

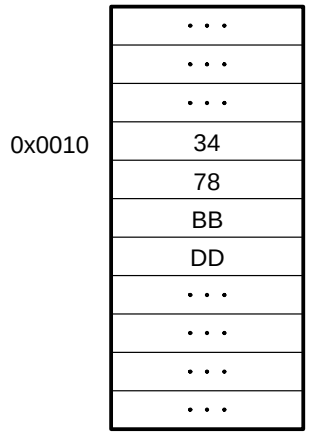
Figure C–2. Data From Output File

(a) low8.bit (Lower Bits)

Data from converted output file

```
:040010003478BBDDA8
:00000001FF
```

Corresponding map in EPROM — ROM0 (See Example C–1 on page C-2)

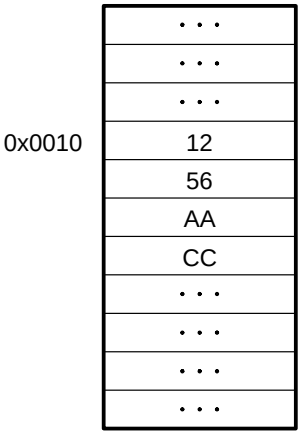


(b) upp8.bit (Upper Bits)

Data from converted output file

```
:040010001256AACC0E
:00000001FF
```

Corresponding map in EPROM — ROM1 (See Example C–1 on page C-2)



To illustrate precisely how the utility performs the conversion, specify the –map option. Although not required, the –map option generates useful information about the output. The resulting map is shown in Example C–4.

Example C–4. Map File Resulting From Hex Command File in Example C–3 on page C-5

```
*****************************************************
TMS320C54x COFF/Hex Converter             Version x.xx
*****************************************************
Fri Oct 11 15:10:53 1996

INPUT FILE NAME: <test.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
   Default data width:    16
   Default memory width:  16
   Default output width: 8

OUTPUT TRANSLATION MAP
-------------------------------------------------------
00000000..0000ffff  Page=0  ROM Width=8  Memory Width=16  ″EPROM″
-------------------------------------------------------
   OUTPUT FILES: low8.bit [b0..b7]
                 upp8.bit [b8..b15]

   CONTENTS: 00000010..00000013  Data Width=2  outsec
```

C.3 Example 2: Avoiding Holes With Multiple Sections

When the memory width is less than the data width, holes may appear at the beginning of a section or between sections. This is due to multiplication of the load address by a correction factor. See Section 12.10, *Controlling the ROM Device Address*, on page 12-34 for more information.

You must eliminate the holes between converted sections. The sections can be made contiguous in one of two ways:

- Specify a paddr value for each section listed in a SECTIONS directive. This forces the hex conversion utility to use that specific address for the output file address field. You must ensure that the section addresses do not overlap. Example C–5 (a) shows a linker command file for this method. The linker should be executed with this command file; then, the hex conversion utility should be executed with the set of commands shown in Example C–5 (b).
- Link the sections together into one output section for conversion. Example C–6 (a) shows a linker command file for this method. The linker should be executed with this command file; then, the hex conversion utility should be executed with the set of commands shown in Example C–6 (b).

Example C–5. Method One for Avoiding Holes

(a) Linker command file

```
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   PAGE 0: DARAM: org = 0x0080  , length = 0x1370
             EXT: org = 0x1400  , length = 0xEB80
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   sec1 : load = EXT PAGE 0
   sec2 : load = EXT PAGE 0
}
```

(b) Hex command file

```
–i
test.out
–map example.mxp

ROMS
{
 PAGE 0: ROM: org = 0x0000,  length = 0x800, romwidth = 8, memwidth = 8
}

SECTIONS
{
  sec1: paddr = 0x0000
  sec2: paddr = 0x0004
}
```

Example C–6. Method Two for Avoiding Holes

(a) Linker command file

```
/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
  PAGE 0: DARAM: org = 0x0080  , length = 0x1370
            EXT: org = 0x1400  , length = 0xEB80
}

SECTIONS
{
   outsec: { *(sec1)
             *(sec2) } > EXT PAGE 0
}
```

(b) Hex command file

```
–i
test.out
–map example.mxp

ROMS
{
  PAGE 0: ROM : org = 0x0100, length = 0x0800, romwidth = 8, memwidth = 8,
                files = {examp2_2.hex}
}

SECTIONS
{
 outsec: paddr = 0x100
}
```

C.4 Example 3: Generating a Boot Table for Non-LP Core Devices

Example 3 shows how to use the linker and the hex conversion utility to build a boot load table for the ’C54x devices. The assembly code used in this section is shown in Example C–1 on page C-2.

Note:

This example is for non-LP ’C54x devices only.

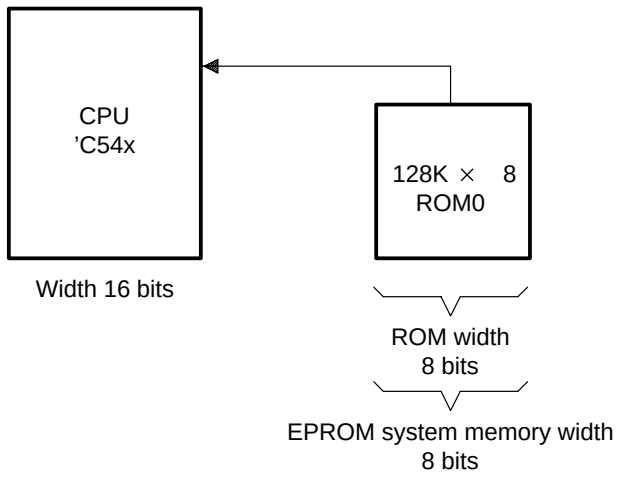
For ’C54xLP devices, see Section C.5, *Example 4: Generating a Boot Table for LP Core Devices*, on page C-17.

Example C–7. C Code for a ’C54x

```
int array[]={1,2,3,4};

main()
{
  array[0] = 5;
}
```

Figure C–3 shows the EPROM memory system for which the output file will be generated. In this application, the single ’C54x device is booted from a 128K × 8-bit EPROM. The requirement of the system is that the boot table must reside at EPROM memory address 0.

Figure C–3. EPROM System for a ’C54x



The on-chip boot loader loads only a single block. This may present a problem when you are loading C code compiled with the TMS320C54x C compiler. The TMS320C54x C compiler creates several sections or blocks when it compiles C source code. Some applications may require that all sections associated with the program be included in the boot to have a complete executable program. In this case, the individual sections must be combined into a single section for boot.

The hex conversion utility does not combine individual sections; therefore, you must use the linker to group those sections.

The sections that the compiler creates are divided into two categories: initialized sections (sections that contain data or code) and uninitialized sections (sections that reserve space but contain no actual data). Initialized sections created by the TMS320C54x C compiler include .text, .cinit, .const, and .data. Uninitialized sections are ignored by the hex conversion utility and are not converted.

Most applications require that .text and .cinit sections are included in the boot. This allows code and information for the C boot routine (c_int00 defined in boot.asm) to load and run, initializing the C environment and branching to the main function in the applications code.

The .text and .cinit sections must be linked together as a single section in the linker command file. The .cinit section contains the initialization data and tables for all global or static C symbols that were declared with an initial value (i.e. int x = 5;). Note that the linker handles the .cinit section differently than the other sections.

When the linker encounters a .cinit section specified as an *output section* in the link, it automatically:

- Sets the symbol cinit to point to the start of the included .cinit section
- Appends a single word to the end of the section

This last word contains a zero that is used to mark the end of the initialization table. However, if .cinit is included as an *input section* only, the linker sets cinit to –1, indicating that no initialization tables were loaded. Therefore, the C boot routine, c_int00, does not attempt to initialize any of the global or static C symbols.

When linking the .cinit section into an output section other than .cinit, the linker does not perform the automatic functions listed above. Therefore, these functions must be implemented explicitly within the linker command file. Example C–8 shows a linker command file that places .text and .cinit into a single output section named boot_sec.

Example C–8. Linker Command File to Form a Single Boot Section for a Non-LP ’C54x

```
–c
–l rts.lib
–m boot1.map
–o boot.out

MEMORY
{
   PAGE 0 : PROG     : origin = 001400h, length = 01000h
   PAGE 1 : DATA     : origin = 0080h,   length = 01000h
}

SECTIONS
{
   boot_sec: { *(.text)
             /*--------------------------------------*/
             /* Set start address for C init table  */
             /*--------------------------------------*/
              cinit = .;
             /*--------------------------------------*/
             /* Include all cinit sections          */
             /*--------------------------------------*/
              *(.cinit)
             /*--------------------------------------*/
             /* Reserve a single space for the zero */
             /* word to mark end of C init          */
             /*--------------------------------------*/
              .+=1;

            }
   fill = 0x0000, /* Make sure fill value is 0   */
   load = PROG PAGE 0
   .data     : {} > DATA   PAGE 1
   .bss      : {} > DATA   PAGE 1
   .const    : {} > DATA   PAGE 1
   .sysmem   : {} > DATA   PAGE 1
   .stack    : {} > DATA   PAGE 1
}
```

Example C–9 shows a portion of the map file generated when the linker is executed with the command file in Example C–8.

Example C–9. Section Allocation Portion of Map File Resulting From the Command File in Example C–8

```
SECTION ALLOCATION MAP

 output                                  attributes/
section    page   origin     length       input sections
--------   ----   ---------- ----------   ----------------
boot_sec   0      00001400   0000006e
                    00001400   00000004    boot.obj (.text)
                    00001404   0000002b    rts.lib  : boot.obj (.text)
                    0000142f   00000035             : exit.obj (.text)
                    00001464   00000006    boot.obj (.cinit)
                    0000146a   00000003    rts.lib : exit.obj (.cinit)
                    0000146d   00000001    --HOLE-- [fill = 0000]

.data      1      00000080   00000000     UNINITIALIZED
                    00000080   00000000    boot.obj (.data)
                    00000080   00000000    rts.lib  : exit.obj (.data)
                    00000080   00000000             : boot.obj (.data)

.bss       1      00000080   00000025     UNINITIALIZED
                    00000080   00000004    boot.obj (.bss)
                    00000084   00000000    rts.lib  : boot.obj (.bss)
                    00000084   00000021             : exit.obj (.bss)

.const     1      00000080   00000000     UNINITIALIZED

.sysmem    1      00000080   00000000     UNINITIALIZED

.stack     1      000000a5   00000400     UNINITIALIZED
                    000000a5   00000000    rts.lib : boot.obj (.stack)

GLOBAL SYMBOLS

address  name                            address  name
-------- ----                            -------- ----
00000080 .bss                            00000080 edata
00000080 .data                           00000080 .data
00000400 __STACK_SIZE                    00000080 _array
0000145e _abort                          00000080 .bss
00000080 _array                          000000a5 end
00001448 _atexit                         00000400 __STACK_SIZE
00001404 _c_int00                        00001400 _main
0000142f _exit                           00001404 _c_int00
00001400 _main                           0000142f _exit
00001464 cinit                           00001448 _atexit
00000080 edata                           0000145e _abort
000000a5 end                             00001464 cinit


[12 symbols]
```

Notice that the linker placed a hole at the end of the section boot_sec with a fill value of 0, as specified in the command file. Also, the global symbol cinit coincides with the start of the first .cinit section included in the link. When the linker is executed with the command file in Example C–8 on page C-12, the linker issues warnings that the output file contains no .text section and that the global symbol cinit is being redefined. These warnings may be ignored in this instance.

Executing the linker with the command file in Example C–8 on page C-12 yields a COFF file that can be used as input to the hex conversion utility to build the desired boot table.

The hex conversion utility has options that describe the requirements for the EPROM programmer and options that describe the EPROM memory system. For Example 3, assume that the EPROM programmer has only one requirement: that the hex file be in Intel format.

In the EPROM memory system illustrated in Figure C–3 on page C-10, the EPROM system memory width is 8 bits, and the physical ROM width is 8 bits. You must set the following options in the hex command file to reflect the requirements of the system:

Option	Description
–i	Create Intel format.
–memwidth 8	Set EPROM system memory width to 8.
–romwidth 8	Set physical ROM width to 8.

Because the application requires the building of a boot table for parallel boot mode, you must set the following options in the hex command file to reflect the requirements of the system:

Option	Description
–boot	Create a boot load table.
–bootorg 0x0000	Place boot table at address 0x0000.

Example C–10. Hex Command File for Converting a COFF File

```
c54x.out            /* Input COFF file               */
–i                  /* Select Intel format           */

–map c54x.mxp

–o c54x.hex         /* Name the hex output file      */

–memwidth 8         /* Set EPROM system memory width */
–romwidth 8         /* Set physical ROM width        */

–boot               /* Make all sections bootable    */
–bootorg 0x0000     /* Place boot table in EPROM     */
                    /* starting at address 0x0000    */



ROMS
{
   PAGE 0 : ROM : origin = 0x0000, length = 0x20000
}
```

In Example 3, memory width and ROM width are the same; therefore, the hex conversion utility creates a single output file. The number of output files is determined by the ratio of memwidth to romwidth.

Example C–11 shows the map file boot.map, resulting from executing the command file in Example C–10, which includes the –map option.

Example C–11. Map File Resulting From the Command File in Example C–10

```
*****************************************************
TMS320C54x COFF/Hex Converter             Version x.xx
*****************************************************
Fri Oct 11 15:27:46 1996

INPUT FILE NAME: <c54x.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
   Default data width:    16
   Default memory width:  8 (MS-->LS)
   Default output width:  8

BOOT LOADER PARAMETERS
   Table Address: 0000, PAGE 0

OUTPUT TRANSLATION MAP
-------------------------------------------------------------------
00000000..0001ffff  Page=0  Memory Width=8  ROM Width=8  "ROM"
-------------------------------------------------------------------
   OUTPUT FILES: c54x.hex [b0..b7]

   CONTENTS: 00000000..000000f7   BOOT TABLE
                                    boot_sec : dest=00001400  size=0000007a
width=00000002
```

The hex conversion utility output file boot.hex, resulting from the command file in Example C–10, is shown in Example C–12.

Example C–12. Hex Conversion Utility Output File Resulting From the Command File in Example C–10

```
:200000001400007976F800800005FC00771800A66BF8001803FF68F80018FFFEF7B8F7BED9
:20002000F4A0F6B7F6B5F6B6F020146DF1000001F84D142BF07314257EF80012F00000010C
:2000400047F800117E9200F80011F00000017EF80011F00000016C89141AF0741400F074CF
:2000600014674A117211008410F80011FA4514444A16EEFF4811F00000868816F495F49527
:2000800010EEFFFFF4E36CE9FFFF143E10F80085F845144B10F80085F4E3F495F073144C0F
:2000A000F7B811F80084F3100020FA4B145BF4954A11F2731465F495E80172110084491198
:2000C00080E10086F3000001E80081F800848A11FC00EEFFE801F074142FEE01FC0000045D
:1800E000008000010002000300040001008400000001008500000000073
:00000001FF
```

C.5 Example 4: Generating a Boot Table for LP Core Devices

Example 4 shows how to use the linker and the hex conversion utility to build a boot load table for the ’C54xLP devices. For the ’C54xLP devices, you can specify multiple sections. It is not necessary, therefore, to group sections at link time as with the non-lp devices. The assembly code used in this section is shown in Example C–1 on page C-2.

Note:

This example is for ’C54xLP devices only.

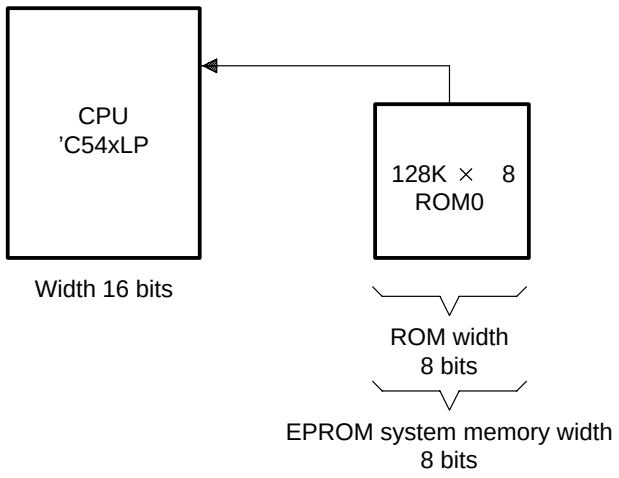
For non-LP ’C54x devices, see Section C.4, *Example 3: Generating a Boot Table for Non-LP Core Devices*, on page C-10.

Example C–13. C Code for a ’C54xLP

```
int array[]={1,2,3,4};

main()
{
  array[0] = 5;
}
```

Figure C–4 shows the EPROM memory system for which the output file will be generated. In this application, the single ’C54xLP device is booted from a 128K × 8-bit EPROM. The requirements of the system are that the boot table must reside at EPROM memory address 0.

Figure C–4. EPROM System for a ’C54xLP



The sections that the compiler creates are divided into two categories: initialized sections (sections that contain data or code) and uninitialized sections (sections that reserve space but contain no actual data). Initialized sections created by the TMS320C54x C compiler include .text, .cinit, .const, and .data. Uninitialized sections are ignored by the hex conversion utility and are not converted.

Most applications require that .text and .cinit sections are included in the boot. This allows code and information for the C boot routine (c_int00 defined in boot.asm) to load and run, initializing the C environment and branching to the main function in the applications code.

The .cinit section contains the initialization data and tables for all global or static C symbols that were declared with an initial value (i.e. int x = 5;). Note that the linker handles the .cinit section differently than the other sections.

When the linker encounters a .cinit section specified as an *output section* in the link, it automatically:

- Sets the symbol cinit to point to the start of the included .cinit section
- Appends a single word to the end of the section

This last word contains a zero that is used to mark the end of the initialization table. However, if .cinit is included as an *input section* only, the linker sets cinit to –1, indicating that no initialization tables were loaded. Therefore, the C boot routine, c_int00, does not attempt to initialize any of the global or static C symbols.

When linking the .cinit section into an output section other than .cinit, the linker does not perform the automatic functions listed above. Therefore, these functions must be implemented explicitly within the linker command file. Example C–14 shows a linker command file for a ’C54xLP device.

Example C–14. Linker Command File for a ’C54xLP

```
–c
c54xlp.obj
–l rts.lib
–m c54xlp.map
–o c54xlp.out

MEMORY
{
   PAGE 0 : PROG     : origin = 001400h, length = 01000h
   PAGE 1 : DATA     : origin = 0080h,   length = 01000h
}

SECTIONS
{
   .text     : {} > PROG   PAGE 0
   .cinit    : {} > PROG   PAGE 0
   .data     : {} > DATA   PAGE 1
   .bss      : {} > DATA   PAGE 1
   .const    : {} > DATA   PAGE 1
   .sysmem   : {} > DATA   PAGE 1
   .stack    : {} > DATA   PAGE 1
}
```

Example C–15 shows the map file generated when the linker is executed with the command file in Example C–14. Linking with this command file creates a COFF file you use as input to the hex conversion utility to build the desired boot table.

Example C–15. Section Allocation Portion of Map File Resulting From the Command File in Example C–14

```
OUTPUT FILE NAME:   <c54xlp.out>
ENTRY POINT SYMBOL: ″_c_int00″  address: 00001404


MEMORY CONFIGURATION

         name     origin    length      used    attributes    fill
        --------  --------  ---------- --------  ---------- --------
PAGE 0: PROG      00001400  000001000 0000007a  RWIX

PAGE 1: DATA      00000080  000001000 00000426  RWIX
```

Example C–15. Section Allocation Portion of Map File Resulting From the Command File in Example C–14 (Continued)

```
SECTION ALLOCATION MAP

output                                    attributes/
section  page   origin         length       input sections
-------- ----   ----------     ----------  ----------------
.text    0      00001400       0000006d
                  00001400       00000004    c54xlp.obj (.text)
                  00001404       0000002b    rts.lib : boot.obj (.text)
                  0000142f       0000003e            : exit.obj (.text)
.cinit   0      0000146d       0000000d
                  0000146d       00000006    c54xlp.obj (.cinit)
                  00001473       00000006    rts.lib : exit.obj (.cinit)
                  00001479       00000001    --HOLE-- [fill = 0000]
.data    1      00000080       00000000    UNINITIALIZED
                  00000080       00000000    c54xlp.obj (.data)
                  00000080       00000000    rts.lib : exit.obj (.data)
                  00000080       00000000            : boot.obj (.data)
.bss     1      00000080       00000026    UNINITIALIZED
                  00000080       00000004    c54xlp.obj (.bss)
                  00000084       00000000    rts.lib : boot.obj (.bss)
                  00000084       00000022            : exit.obj (.bss)
.const   1      00000080       00000000    UNINITIALIZED
.sysmem  1      00000080       00000000    UNINITIALIZED
.stack   1      000000a6       00000400    UNINITIALIZED
                  000000a6       00000000    rts.lib : boot.obj (.stack)

GLOBAL SYMBOLS

address  name                             address   name
-------- ----                             --------  ----
00000080 .bss                             00000001  __lflags
00000080 .data                            00000080  _array
00001400 .text                            00000080  .data
0000144b C$$EXIT                          00000080  .bss
00000400 __STACK_SIZE                     00000080  edata
00000085 __cleanup_ptr                    00000085  __cleanup_ptr
00000001 __lflags                         000000a6  end
00001467 _abort                           00000400  __STACK_SIZE
00000080 _array                           00001400  .text
0000144e _atexit                          00001400  _main
00001404 _c_int00                         00001404  _c_int00
0000142f _exit                            0000142f  _exit
00001400 _main                            0000144b  C$$EXIT
0000146d cinit                            0000144e  _atexit
00000080 edata                            00001467  _abort
000000a6 end                              0000146d  etext
0000146d etext                            0000146d  cinit
ffffffff pinit                            ffffffff  pinit

[18 symbols]
```

The hex conversion utility has options that describe the requirements for the EPROM programmer and options that describe the EPROM memory system. For Example 4, assume that the EPROM programmer has only one requirement: that the hex file be in Intel format.

In the EPROM memory system illustrated in Figure C–4 on page C-17, the EPROM system memory width is 8 bits and the physical ROM width is 8 bits. The following options are selected to reflect the requirements of the system:

Option	Description
–i	Create Intel format.
–memwidth 8	Set EPROM system memory width to 8.
–romwidth 8	Set physical ROM width to 8.

Because the application requires the building of a boot table for parallel boot mode, the following options must be selected as well:

Option	Description
–boot	Create a boot load table.
–bootorg 0x0000	Place boot table at address 0x0000.

Example C–16. Hex Command File for Converting a COFF File

```
c54xlp.out          /* Input COFF file                  */
–i                  /* Select Intel format              */

–map c54xlp.mxp     /* Name hex utility map file        */

–o c54xlp.hex       /* Name the hex output file         */

–memwidth 8         /* Set EPROM system memory width    */
–romwidth 8         /* Set physical ROM width           */

–boot               /* Make all sections bootable       */
–bootorg 0x0000     /* Place boot table in EPROM        */
                    /* starting at address 0x0000       */



ROMS
{
   PAGE 0 : ROM : origin = 0x0000, length = 0x20000
}
```

In Example 4, memory width and ROM width are the same; therefore, the hex conversion utility creates a single output file. The number of output files is determined by the ratio of memwidth to romwidth.

Example C–17 shows the map file c54xlp.mxp, resulting from executing the command file in Example C–16, which includes the –map option.

Example C–17. Map File Resulting From the Command File in Example C–16

```
*****************************************************
TMS320C54x COFF/Hex Converter             Version 1.20
*****************************************************
Sat Sep 21 17:01:13 1996

INPUT FILE NAME: <c54xlp.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
   Default data width:    16
   Default memory width:  8 (MS-->LS)
   Default output width:  8

BOOT LOADER PARAMETERS
   Table Address: 0000, PAGE 0


OUTPUT TRANSLATION MAP
------------------------------------------------------------------------
00000000..0001ffff  Page=0  Memory Width=8  ROM Width=8  "ROM"
------------------------------------------------------------------------
   OUTPUT FILES: c54xlp.hex [b0..b7]

   CONTENTS: 00000000..00000109   BOOT TABLE
                                    .text : dest=00001400  size=0000006d
width=00000002
                                    .cinit : dest=0000146d  size=0000000d
width=00000002
```

The hex conversion utility output file c54xlp.hex, resulting from the command file in Example C–16, is shown in Example C–18.

Example C–18. Hex Conversion Utility Output File Resulting From the Command File in Example C–16

```
:2000000008AA7FFFF800FFFFFFFF006C0000140076F800800005FC00771800A66BF800189D
:2000200003FF68F80018FFFEF7B8F7BEF4A0F6B7F6B5F6B6F020146DF1000001F84D142B46
:20004000F07314257EF80012F000000147F800117E9200F80011F00000017EF80011F000BA
:2000600000016C89141AF0741400F07414674A117211008410F80011FA4514444A16EEFFA6
:200080004811F00000868816F495F49510EEFFFFF4E36CE9FFFF143E10F80085F845144B40
:2000A00010F80085F4E3F495F073144CF7B811F80084F3100020FA4B145BF4954A11F27334
:2000C0001465F495E80172110084491180E10086F3000001E80081F800848A11FC00EEFF90
:0A00E000E801F074142FEE01FC009B
:2000EA00000C0000146D000400800001000200030004000100840000000100850000000D0
:00000001FF
```

Appendix D

Error Messages

This appendix lists the assembler and linker error messages in numeric and alphabetical order. Some messages have error-type numbers associated with them. These messages are listed first. If an error-type has multiple messages, the messages are in alphabetical order. The messages without numbers are listed in alphabetical order.

Most error messages have a *Description* of the problem and an *Action* that suggests possible remedies. Where the error message itself is an adequate description, you may find only the *Action* suggested. Where the *Action* is obvious from the description (to inspect and correct your code), the *Action* is omitted.

When the assembler completes its second pass, it reports any errors that it encountered during the assembly. It also prints these errors in the listing file (if one is created); an error is printed following the source line that incurred it. Most assembler errors are fatal errors. If an error is not fatal or if it is a macro error, this is noted in the assembler listing file. The assembler issues the following types of error messages:

- Fatal
- Nonfatal
- Macro

In the linker messages, the symbol (...) represents the name of an object that the linker is attempting to interact with when an error occurs. The linker issues the following types of error messages:

- Syntax and command errors
- Allocation errors
- I/O errors

E0000

Attempt to nest repeat block
Comma required to separate arguments
Illegal combination of shift operands
Illegal instruction
Illegal repeat block open – check delay slot
Illegal repeat block open – missing 'repeat'
Illegal shift for parallel operation
Left parenthesis expected
Matching right parenthesis is missing
Missing opening brace
Missing right quote of string constant
No matching right parenthesis
Open repeat block at EOF
Right parenthesis expected
Syntax Error
Syntax requires parentheses
Unrecognized character type
Unrecognized special character

Description These are errors about general syntax. The required syntax is not present.

Action Correct the source per the error message text.

E0001

Section *sym* is not an initialized section
Section *sym* is not defined

Description These are errors about invalid symbol names. A symbol is invalid for the context in which it is used.

Action Correct the source per the error message text.

E0002

Invalid directive specification
Invalid mnemonic specification

Description These errors are about invalid mnemonics. The instruction, macro, or directive specified was not recognized.

Action Check the directive or instruction used.

Invalid instruction for specified processor version

Description The indicated instruction is not allowed for the processor version specified with the .version directive.

Action Check the instruction and .version directive.

E0003

Cluttered character operand encountered
Cluttered string constant operand encountered
Cluttered identifier operand encountered
Condition must be EQ, LT, GT, or NEQ
Condition must be srcLT, LEQ, GT, GEQ
Condition must be srcEQ, NEQ, LT, LEQ, GT, or GEQ
Expecting ARn for src,dst
Expecting shift or accumulator
Illegal auxiliary register specified
Illegal condition operand
Illegal condition operand or combination
Illegal indirect memaddr specification
Illegal operand
Illegal smem operand
Immediate value out of range
Incorrect bit symbol for specified status register
Invalid binary constant specified
Invalid constant specification
Invalid decimal constant specified
Invalid float constant specified

Invalid hex constant specified
Invalid immediate expression or shift value
Invalid immediate or shift value
Invalid octal constant specified
Invalid Operand 2
Invalid operand *x*
Invalid shift value
Must add AR0 to destination
Must subtract AR0 from destination");
Operand must be the A accumulator
Operand must be the B accumulator
Shift value must be 16
Shift value out of range
Syntax error – Operand nnn
The accumulator arguments must be the same
The dst accumulator arguments must be the same
The dst,src1 arguments must be the same
The smem operands must be the same

Description These are errors about invalid operands. The instruction, parameter, or other operand specified was not recognized.

Action Correct the source per the error message text.

E0004

Absolute, well-defined integer value expected
Accumulator specified in second half of parallel instruction may not be the same as the first
Data size must be equal to pointer size
Expecting accumulator A or B
Expecting ASM or shift value
Expecting dual memory addressing
Identifier operand expected
Illegal character argument specified
Illegal combination of Smem operands
Illegal endian flag specified
Illegal floating-point expression
Illegal operand
Illegal shift operation
Illegal string constant operand specified
Illegal structure reference

Incorrect bit symbol fdor specified status register
Invalid data size for relocation
Invalid float constant specified
Invalid identifier, *sym*, specified
Invalid macro parameter specified
Invalid operand, "char"
Must add to the destination operand
No parameters available for macro arguments
Not expecting direct operand *op*
Not expecting indirect operand *op*
Not expecting immediate value operand *op*
Operand must be auxiliary register or SP
Operand must be auxiliary register
Operand must be T
Offset Addressing modes not legal for MMRs
Pointer too big for this data size
Register must be ARn or SP
Single character operand expected
String constant or substitution symbol expected
String operand expected
Structure/Union tag symbol expected
Substitution symbol operand expected
The accumulator operands must be different
The operands must be SP

Description These errors are about illegal operands. The instruction, parameter or other operand specified was not legal for this syntax.

Action Correct the source per the error message text.

E0005

Missing field value operand
Missing operand(s)

Description These are errors about missing operands; a required operand is not supplied.

Action Correct the source so that all required operands are declared.

E0006

.break must occur within a loop
Conditional assembly mismatch
Matching .endloop missing
Matching .macro missing
No matching .if specified
No matching .endif specified
No matching .endloop specified
No matching .loop specified
Open block(s) inside macro
Unmatched .endloop directive
Unmatched .if directive

Description These are errors about unmatched conditional assembly directives. A directive was encountered that requires a matching directive but the assembler could not find the matching directive.

Action Correct the source per the error message text.

E0007

Conditional nesting is too deep
Loop count out of range

Description These are errors about conditional assembly loops. Conditional block nesting cannot exceed 32 levels.

Action Correct the .macro/.endmacro, .if/.elseif/.else/.endif or .loop/.break/.endloop source.

E0008

Bad use of .access directive
Matching .struct directive is not present
Matching .union directive is not present

Description These are errors about unmatched structure definition directives. In a .struct/.endstruct sequence, a directive was encountered that requires a matching directive but the assembler could not find the matching directive.

Action Check the source for mismatched structure definition directives and correct.

E0009

Cannot apply bitwise NOT to floats
Illegal struct/union reference dot operator
Missing structure tag
Section ”name” is not an initialized section
Structure or union tag symbol expected
Structure or union tag symbol not found
Unary operator must be applied to a constant

Description These are errors about an illegally used operator. The operator specified was not legal for the given operands.

Action Correct the source per the error message text so that all required operands are declared.

E0100

Label missing
.setsym requires a label

Description These are errors about required labels. The given directive requires a label, but none is specified.

Action Correct the source by specifying the required label.

E0101

Labels are not allowed with this directive
Standalone labels not permitted in structure/union defs

Description These are errors about invalid labels. The given directive does not permit a label, but one is specified.

Description Remove the invalid label.

E0102

Local label *number* defined differently in each pass
Local label *number* is multiply defined
Local label *number* is not defined in this section
Local labels can't be used with directives
The accumulator operands must be different

Description These are errors about the illegal use of local labels.

Action Correct the source per the error message text. Use .newblock to reuse local labels.

E0200

Bad term in expression
Binary operator can't be applied
Difference between segment symbols not permitted
Divide by zero
Division by zero is illegal
Expression evaluation failed
Expression must be absolute integer value
Offset expression must be integer value
Operation cannot be performed on given operands
Unary operator can't be applied
Value of expression has changed due to jump expansion
Well–defined expression required

Description These are errors about general expressions. An illegal operand combination was used, or an arithmetic type is required but not present.

Action Correct the source per the error message text.

E0201

Absolute operands required for FP operations!
Cannot apply bitwise NOT to floats
Floating–point divide by zero
Floating–point overflow
Floating–point underflow
Floating–point expression required
Illegal floating–point expression
Invalid floating–point operation

Description These are errors about floating-point expressions. A floating-point expression was used where an integer expression is required, an integer expression was used where a floating-point expression is required, or a floating-point value is invalid.

Action Correct the source per the error message text.

E0300

Cannot equate an external symbol to an external
Cannot redefine this section name
Cannot tag an undefined symbol
Empty structure or union definition
Illegal structure or union tag
Missing closing '}' for repeat block
Redefinition of "sym" attempted
Structure member previously defined
Structure tag can't be global
Symbol can't be defined in terms of itself
Symbol expected
Symbol expected in label field
Symbol, *sym*, has already been defined
Symbol, *sym*, is not defined in this source file
Symbol, *sym*, is operand to both .ref and .def
Structure/union member, *sym*, not found
The following symbols are undefined:

Union member previously defined
Union tag can't be global

Description These are errors about general symbols. An attempt was made to redefine a symbol or to define a symbol illegally.

Action Correct the source per the error message text.

E0301

Cannot redefine local substitution symbol
Substitution Stack Overflow
Substitution symbol not found

Description These are errors about general substitution symbols. An attempt was made to redefine a symbol or to define a symbol illegally.

Action Correct the source per the error message text. Make sure that the operand of a substitution symbol is defined either as a macro parameter or with a .asg or .eval directive.

E0400

Symbol table entry is not balanced

Description A symbolic debugging directive does not have a complementing directive (i.e., a .block without an .endblock).

Action Check the source for mismatched conditional assembly directives.

E0500

Macro argument string is too long
Missing macro name
Too many variables declared in macro

Description These are errors about general macros. A macro definition was probably corrupted.

Action Correct the source per the error message text.

E0501

Macro definition not terminated with .endm
Matching .endm missing
Matching .macro missing
.mexit directive outside macro definition
No active macro definition

Description These are errors about macro definition directives. A macro directive does not have a complementing directive (that is, a .macro without a .endm).

Action Correct the source per the error message text.

E0600

Bad archive entry for *macro name*
Bad archive name
Can't read a line from archive entry
***library name* macro library not found**
***library name* is not in archive format**

Description These are errors about macro library accessing. A problem was encountered reading from or writing to a macro library archive file. It is likely that the creation of the archive file was not done properly.

Action Make sure that the macro libraries are unassembled assembler source files. Also make sure that the macro name and member name are the same, and the extension of the file is .asm.

E0700

Illegal structure/union member
No structure/union currently open
.sym not allowed inside structure/union

Description These are errors about the illegal use of symbolic debugging directives; a symbolic debugging directive is not used in an appropriate place.

Action Correct the source per the error message text.

E0800

Delayed branch – too many words in delay slot
Delayed branch – control flow in delay slot

Description These are errors about branch instructions. These errors are normally target specific.

Action Correct the source per the error message text.

E0801

Instructions not permitted in structure/union definitions

Description An invalid instruction was encountered in a structure or union definition.

Action Correct the source by removing the invalid instruction(s).

E0802

Expecting parallel instruction
Incorrect instruction used in parallel
Illegal form of LD used in parallel

Description These are errors about illegal used parallel instructions.

Action Correct the source per the error message text.

E0900

Can’t include a file inside a loop or macro
Illegal structure member
Illegal structure definition contents
Invalid load–time label
Invalid structure/union contents

.setsect only valid if absolute listing produced (use –a option)
.setsym only valid if absolute listing produced (use –a option)
.var allowed only within macro definitions

Description These are errors about illegally used directives. Specific directives were encountered where they are not permitted because they will cause a corruption of the object file. Many directives are not permitted inside of structure or union definitions.

Action Correct the source per the error message text.

E1000

Include/Copy file not found or opened

Description The specified filename cannot be found.

Action Check spelling, pathname, environment variables, etc.

E1300

Copy limit has been reached
Exceeded limit for macro arguments
Macro nesting limit exceeded

Description These errors are about general assembler limits that have been exceeded. The nesting of .copy/.include files in limited to 10 levels. Macro arguments are limited to 32 parameter. Macro nesting is limited to 32 levels.

Action Check the source to determine how limits have been exceeded.

W0000

No operands expected. Operands ignored
Trailing operands ignored
***+ARn addressing is for write-only**

Description These are warnings about operands. The assembler encountered operands that it did not expect.

Action Check the source to determine what caused the problem and whether you need to correct the source.

W0001

Field value truncated to *value*
Field width truncated to *size in bits*
Line too long, will be truncated
Power of 2 required, *next larger power of 2* assumed
Section Name is limited to 8 characters
String is too long – will be truncated
Value truncated
Value truncated to byte size
Value out of range

Description These are warnings about truncated values. The expression given was too large to fit within the instruction opcode or the required number of bits.

Action Check the source to make sure the result will be acceptable, or change the source if an error has occurred.

W0002

Address expression will wrap–around
Expression will overflow, value truncated

Description These are warnings about arithmetic expressions. The assembler has done a calculation that will produce the indicated result, which may or may not be acceptable.

Action Verify the result will be acceptable, or change the source if an error has occurred.

W0003

.sym for *function name* required before .func

Description This is a warning about problems with symbolic debugging directives. A .sym directive defining the function does not appear before the .func directive.

Action Correct the source.

A

absolute symbol (...) being redefined

Description An absolute symbol cannot be redefined.

Action Check the syntax of all expressions, and check the input directives for accuracy.

adding name (...) to multiple output sections

Description The input section is mentioned twice in the SECTIONS directive.

ALIGN illegal in this context

Description Alignment of a symbol is performed outside of a SECTIONS directive.

alignment for (...) must be a power of 2

Description Section alignment was not a power of 2.

Action Make sure that in hexadecimal, all powers of 2 consist of the integers 1, 2, 4, or 8 followed by a series of zero or more 0s.

alignment for (...) redefined

Description More than one alignment is supplied for a section.

attempt to decrement DOT

Description A statement such as **.–= *value*** is supplied; this is illegal. Assignments to **dot** can be used only to create holes.

B

bad fill value

Description The fill value must be a 16-bit constant.

binding address (...) for section (...) is outside all memory on page (...)

Description Not every section falls within memory configured with the MEMORY directive.

Action If you are using a linker command file, check that MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

binding address (...) for section (...) overlays (...) at (...)

Description Two sections overlap and cannot be allocated.

Action If you are using a linker command file, check that MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

binding address for (...) redefined

Description More than one binding value is supplied for a section.

binding address (...) incompatible with alignment for section (...)

Description The section has an alignment requirement from an .align directive or previous link. The binding address violates this requirement.

blocking for (...) must be a power of 2

Description Section blocking is not a power of 2

Action Make sure that in hexadecimal, all powers of 2 consist of the integers 1, 2, 4, or 8 followed by a series of zero or more 0s.

blocking for (...) redefined

Description More than one blocking value is supplied for a section.

C

–c requires fill value of 0 in .cinit (... overridden)

Description The .cinit tables must be terminated with 0, therefore, the fill value of the .cinit section must be 0.

cannot complete output file (...), write error

Description This usually means that the file system is out of space.

cannot create output file (...)

Description This usually indicates an illegal filename.

Action Check spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

cannot resize (...), section has initialized definition in (...)

Description An *initialized* input section named .stack or .heap exists, preventing the linker from resizing the section.

cannot specify a page for a section within a GROUP

Description A section was specified to a specific page within a group. The entire group is treated as one unit, so the group may be specified to a page of memory, but the sections making up the group cannot be handled individually.

cannot specify both binding and memory area for (...)

Description Both binding and memory were specified. The two are mutually exclusive.

Action If you wish the code to be placed at a specific address, use binding only.

can't align a section within GROUP – (...) not aligned

Description A section in a group was specified for individual alignment. The entire group is treated as one unit, so the group may be aligned or bound to an address, but the sections making up the group cannot be handled individually.

can't align within UNION – section (...) not aligned

Description A section in a union was specified for individual alignment. The entire union is treated as one unit, so the union may be aligned or bound to an address, but the sections making up the union cannot be handled individually.

can't allocate (...), size ... (page ...)

Description A section can't be allocated, because no existing configured memory area is large enough to hold it.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

can’t create map file (...)

Description Usually indicates an illegal filename.

Action Check spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t find input file *filename*

Description The file, *filename*, is not in your PATH, is misspelled, etc.

Action Check spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t open (...)

Description The specified file does not exist.

Action Check spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t open *filename*

Description The specified file does not exist.

Action Check spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t read (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

can’t seek (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

can’t write (...)

Description Disk may be full or protected.

Action Check disk volume and protection.

command file nesting exceeded with file (...)

Description Command file nesting is allowed up to 16 levels.

E

–e flag does not specify a legal symbol name (...)

Description The –e option is not supplied with a valid symbol name as an operand.

entry point other than _c_int00 specified

Description For –c or –cr option only. A program entry point other than the value of _c_int00 was supplied. The runtime conventions of the compiler assume that _c_int00 is the one and only entry point.

entry point symbol (...) undefined

Description The symbol used with the –e option is not defined.

errors in input – (...) not built

Description Previous errors prevent the creation of an output file.

F

fail to copy (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

fail to read (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

fail to seek (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

fail to skip (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

fail to write (...)

Description The disk may be full or protected.

Action Check disk volume and protection.

file (...) has no relocation information

Description You have attempted to relink a file that was not linked with –r.

file (...) is of unknown type, magic number = (...)

Description The binary input file is not a COFF file.

fill value for (...) redefined

Description More than one fill value is supplied for an output section. Individual holes can be filled with different values with the section definition.

I

–i path too long (...)

Description The maximum number of characters in an –i path is 256.

illegal input character

Description There is a control character or other unrecognized character in the command file.

illegal memory attributes for (...)

Description The attributes are not some combination of R, W, I, and X.

illegal operator in expression

Description Review legal expression operators.

illegal option within SECTIONS

Description The –l (lowercase L) option is the only option allowed within a SECTIONS directive.

illegal relocation type (...) found in section(s) of file (...)

Description The binary file is corrupt.

internal error (...)

Description This linker has an internal error.

invalid archive size for file (...)

Description The archive file is corrupt.

invalid path specified with –i flag

Description The operand of the –i option (flag) is not a valid file or pathname.

invalid value for –f flag

Description The value for –f option (flag) is not a 2-byte constant.

invalid value for –heap flag

Description The value for –heap option (flag) is not a 2-byte constant.

invalid value for –stack flag

Description The value for –stack option (flag) is not a 2-byte constant.

invalid value for –v flag

Description The value for –v option (flag) is not a constant.

I/O error on output file (...)

Description The disk may be full or protected.

Action Check disk volume and protection.

L

length redefined for memory area (...)

Description A memory area in a MEMORY directive has more than one length.

library (...) member (...) has no relocation information

Description The library member has no relocation information. It is possible for a library member to not have relocation information; this means that it cannot satisfy unresolved references in other files when linking.

line number entry found for absolute symbol

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

load address for uninitialized section (...) ignored

Description A load address is supplied for an uninitialized section. Uninitialized sections have no load addresses—only run addresses.

load address for UNION ignored

Description UNION refers only to the section's run address.

load allocation required for initialized UNION member (...)

Description A load address is supplied for an initialized section in a union. UNIONs refer to runtime allocation only. You must specify the load address for all sections within a union separately.

M

–m flag does not specify a valid filename

Description You did not specify a valid filename for the file you are writing the output map file to.

making aux entry *filename* for symbol *n* out of sequence

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

memory area for (...) redefined

Description More than one named memory allocation is supplied for an output section.

memory page for (...) redefined

Description More than one page allocation is supplied for a section.

memory attributes redefined for (...)

Description More than one set of memory attributes is supplied for an output section.

memory types (...) and (...) on page (...) overlap

Description Memory ranges on the same page overlap.

Action If you are using a linker command file, check that MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

missing filename on –l; use –l <filename>

Description No filename operand is supplied for the –l (lowercase L) option.

misuse of DOT symbol in assignment instruction

Description The "." symbol is used in an assignment statement that is outside the SECTIONS directive.

N

no allocation allowed for uninitialized UNION member

Description A load address was supplied for an uninitialized section in a union. An uninitialized section in a union gets its run address from the UNION statement and has no load address, so no load allocation is valid for the member.

no allocation allowed with a GROUP–allocation for section (...) ignored

Description A section in a group was specified for individual allocation. The entire group is treated as one unit, so the group may be aligned or bound to an address, but the sections making up the group cannot be handled individually.

no input files

Description No COFF files were supplied. The linker cannot operate without at least one input COFF file.

no load address specified for (...); using run address

Description No load address is supplied for an initialized section. If an initialized section has a run address only, the section is allocated to run and load at the same address.

no run allocation allowed for union member (...)

Description A UNION defines the run address for all of its members; therefore, individual run allocations are illegal.

no string table in file *filename*

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

no symbol map produced – not enough memory

Description Available memory is insufficient to produce the symbol list. This is a nonfatal condition that prevents the generation of the symbol list in the map file.

O

–o flag does not specify a valid file name : *string*

Description The filename must follow the operating system file naming conventions.

origin missing for memory area (...)

Description An origin is not specified with the MEMORY directive. An origin specifies the starting address of a memory range.

out of memory, aborting

Description Your system does not have enough memory to perform all required tasks.

Action Try breaking the assembly language files into multiple smaller files and do partial linking. See Section 9.16, *Partial (Incremental) Linking*, on page 9-63.

output file has no .bss section

Description This is a warning. The .bss section is usually present in a COFF file. There is no requirement for it to be present.

output file has no .data section

Description This is a warning. The .data section is usually present in a COFF file. There is no requirement for it to be present.

output file has no .text section

Description This is a warning. The .text section is usually present in a COFF file. There is no requirement for it to be present.

output file (...) not executable

Description The output file created may have unresolved symbols or other problems stemming from other errors. This condition is not fatal.

overwriting aux entry *filename* of symbol *n*

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

P

PC-relative displacement overflow at address (...) in file (...)

Description The relocation of a PC-relative jump resulted in a jump displacement too large to encode in the instruction.

R

–r incompatible with –s (–s ignored)

Description Both the –r option and the –s option were used. Since the –s option strips the relocation information and –r requests a relocatable object file, these options are in conflict with each other.

relocation entries out of order in section (...) of file (...)

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

relocation symbol not found: index (...), section (...), file (...)

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

S

section (...) at (...) overlays at address (...)

Description Two sections overlap and cannot be allocated.

Action If you are using a linker command file, check that MEMORY and SECTIONS directives allow enough room to ensure that no sections overlap.

section (...) enters unconfigured memory at address (...)

Description A section can't be allocated because no existing configured memory area is large enough to hold it.

Action If you are using a linker command file, check that MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

section (...) not built

Description Most likely there is a syntax error in the SECTIONS directive.

section (...) not found

Description An input section specified in a SECTIONS directive was not found in the input file.

section (...) won't fit into configured memory

Description A section can't be allocated, because no configured memory area exists that is large enough to hold it.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

seek to (...) failed

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

semicolon required after assignment

Description There is a syntax error in the command file.

statement ignored

Description There is a syntax error in an expression.

symbol referencing errors — (...) not built

Description Symbol references could not be resolved. Therefore, an object module could not be built.

symbol (...) from file (...) being redefined

Description A defined symbol is redefined in an assignment statement.

T

too few symbol names in string table for archive *n*

Description The archive file may be corrupt.

Action If the input file is corrupt, try recreating the archive.

too many arguments – use a command file

Description You used more than ten arguments on a command line or in response to prompts.

too many –i options, 7 allowed

Action More than seven –i options were used. Additional search directories can be specified with a C_DIR or A_DIR environment variable.

type flags for (...) redefined

Description More than one section type is supplied for a section. Note that type COPY has all of the attributes of type DSECT, so DSECT need not be specified separately.

type flags not allowed for GROUP or UNION

Description A type is specified for a section in a group or union. Special section types apply to individual sections only.

U

–u does not specify a legal symbol name

Description The –u option did not specify a legal symbol name that exists in one of the files that you are linking.

unexpected EOF(end of file)

Description There is a syntax error in the linker command file.

undefined symbol (...) first referenced in file (...)

Description Either a referenced symbol is not defined, or the –r option was not used. Unless the –r option is used, the linker requires that all referenced symbols be defined. This condition prevents the creation of an executable output file.

Action Link using the –r option or define the symbol.

undefined symbol in expression

Description An assignment statement contains an undefined symbol.

unrecognized option (...)

Action Check the list of valid options.

Z

zero or missing length for memory area (...)

Description A memory range defined with the MEMORY directive did not have a nonzero length.

Appendix E

Glossary

A

absolute address: An address that is permanently assigned to a TMS320C54x 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.

algebraic: An instruction that the assembler translates into machine code.

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 alignment 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.

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.

COFF: Common object file format. A binary object file format that promotes modular programming by supporting the concept of *sections.*

command file: A file that contains options, filenames, directives, or comments 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: See *COFF*.

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 can be used as an operand.

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

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 TMS320C54x 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 TMS320C54x 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 TMS320C54x, a software-configurable data type whose length can be programmed to be any value in the range of 1–16 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: 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 that accepts COFF files and converts them into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer.

high-level language debugging: The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

hole: An area between the input sections that compose an output section that contains no actual code or data.

I

incremental linking: Linking files that will be linked in several passes. Often this means a very large file that will have sections linked and then will have the sections linked 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 TMS320C54x 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.

loader: A device that loads an executable module into TMS320C54x system memory.

M

member: The elements or variables of a structure, union, archive, or enumeration.

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 TMS320C54x.

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.

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 format converter: A program that converts COFF object files into Intel format or Tektronix format object files.

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.

overlay page: A section of physical memory that is mapped into the same address range as another section of memory. A hardware switch determines which range is active.

P

partial linking: The linking of a file that will be linked again.

Q

quiet run: Suppresses the normal banner and the progress information.

R

RAM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the –cr option. The RAM model allows variables to be initialized at load time instead of runtime.

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 model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the –c option. In the ROM model, the linker loads the .cinit section of data tables into memory, and variables are initialized at runtime.

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 TMS320C54x 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 TMS320C54x 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 of the assembler that keeps track of the current location within a section; each section has its own SPC.

static: A kind of 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; their previous value is resumed when the function or program is re-entered.

storage class: Any entry in the symbol table that indicates how to access a symbol.

string table: A table that stores symbol names that are longer than 8 characters (symbol names of 8 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 smaller section within a section offering tighter control of the memory map. See also *section*.

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 TMS320C54x 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 may 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: An expression that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

word: A 16-bit addressable location in target memory.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.