TMS320C55x Assembly Language Tools v 4.4

User's Guide

Literature Number: SPRU280I
November 2011

Contents

List of Figures

List of Tables

Read This First

About This Manual

The *TMS320C55x Assembly Language Tools User's Guide* explains how to use these assembly language tools:

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

How to Use This Manual

This book helps you learn how to use the Texas Instruments assembly language tools designed specifically for the TMS320C55x™ 24-bit devices. This book consists of four parts:

- **Introductory information**, consisting of Chapter 1 and Chapter 2, gives you an overview of the assembly language development tools. It also discusses object modules, which helps you to use the TMS320C55x tools more effectively. Read Chapter 2 before using the assembler and linker.
- **Assembler description**, consisting of Chapter 3 through Chapter 5, contains detailed information about using the assembler. This portion explains how to invoke the assembler and discusses source statement format, valid constants and expressions, assembler output, and assembler directives. It also describes the macro language.
- **Additional assembly language tools description**, consisting of Chapter 8 through Chapter 13, describes in detail each of the tools provided with the assembler to help you create executable object files. For example, Chapter 9 explains how to invoke the linker, how the linker operates, and how to use linker directives; Chapter 13 explains how to use the hex conversion utility.
- **Reference material**, consisting of Appendix A through Appendix C, provides supplementary information including symbolic debugging directives that the TMS320C55x C/C++ compiler uses. It also provides a description of the XML link information file and a glossary.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`. Interactive displays use a bold version of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample of C code:

```
#include <stdio.h>
main()
{    printf("hello, cruel world\n");
}
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered.
- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in the **bold typeface**, do not enter the brackets themselves. The following is an example of a command that has an optional parameter:

cl55 [*options*] [*filenames*] [**--run_linker** [*link_options*] [*object files*]]

- Braces ({ and }) indicate that you must choose one of the parameters within the braces; you do not enter the braces themselves. This is an example of a command with braces that are not included in the actual syntax but indicate that you must specify either the --rom_model or --ram_model option:

cl55 --run_linker {**--rom_model | --ram_model**} *filenames* [**--output_file=** *name.out*]
--library= *libraryname*

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

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

- Some directives can have a varying number of parameters. For example, the .byte directive can have multiple parameters. This syntax is shown as [, ..., *parameter*].
- The TMS320C55x devices are referred to as C55x.
- Following are other symbols and abbreviations used throughout this document:

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

Related Documentation From Texas Instruments

You can use the following books to supplement this user's guide:

SPRAAO8 — **Common Object File Format Application Report.** Provides supplementary information on the internal format of COFF object files. Much of this information pertains to the symbolic debugging information that is produced by the C compiler.

SPRU281 —**TMS320C55x Optimizing C Compiler User's Guide**. Describes the TMS320C55x C/C++ Compiler. This C/C++ compiler accepts ISO standard C/C++ source code and produces assembly language source code for TMS320C55x devices.

SPRU317 —**TMS320C55x DSP Peripherals Reference Guide.** Introduces the peripherals, interfaces, and related hardware that are available on TMS320C55x DSPs.

SPRU376 —**TMS320C55x DSP Programmer's Guide.** Describes ways to optimize C and assembly code for the TMS320C55x™ DSPs and explains how to write code that uses special features and instructions of the DSP.

SPRU393 —**TMS320C55x Technical Overview.** Introduces the TMS320C55x digital signal processor (DSP). The TMS320C55x is the latest generation of fixed-point DSPs in the TMS320C5000™ DSP platform. Like the previous generations, this processor is optimized for high performance and low-power operation. This book describes the CPU architecture, low-power enhancements, and embedded emulation features of the TMS320C55x.

SWPU067 —**TMS320C55x v3.x CPU Mnemonic Instruction Set Reference Guide.** Describes the TMS320C55x 3.x DSP mnemonic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

SWPU068 —**TMS320C55x v3.x CPU Algebraic Instruction Set Reference Guide.** Describes the TMS320C55x 3.x DSP algebraic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

SWPU073 —**TMS320C55x DSP CPU Reference Guide, Version 3.0.** Describes the architecture, registers, and operation of the CPU for the TMS320C55x™ digital signal processors (DSPs).

Introduction to the Software Development Tools

The TMS320C55x™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities. This chapter provides an overview of these tools.

The TMS320C55x is supported by the following assembly language development tools:

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

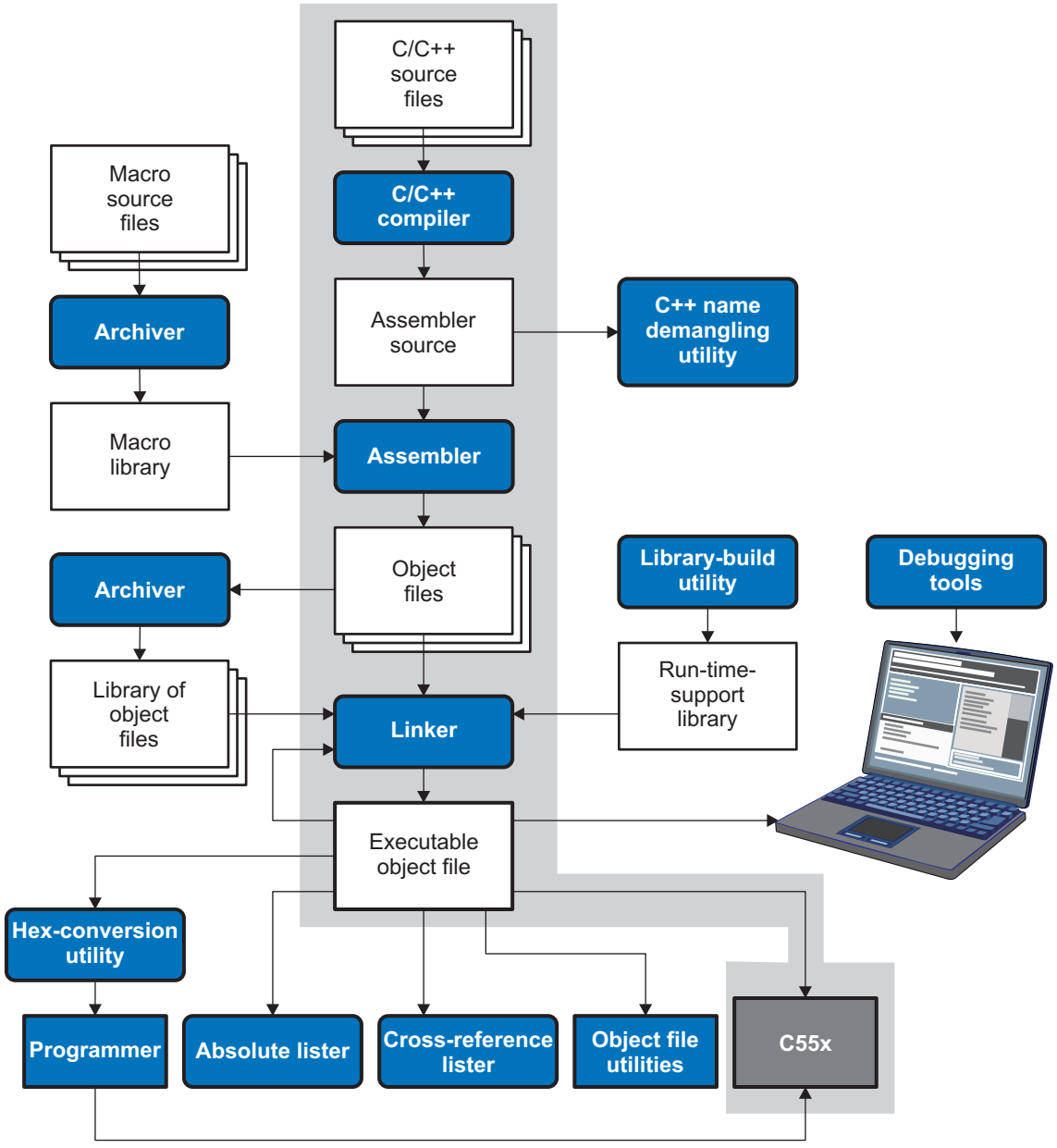
This chapter shows how these tools fit into the general software tools development flow and gives a brief description of each tool. For convenience, it also summarizes the C/C++ compiler and debugging tools. For detailed information on the compiler and debugger, and for complete descriptions of the TMS320C55x, refer to the books listed in *Related Documentation From Texas Instruments*.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1-1 shows the TMS320C55x software development flow. The shaded portion highlights the most common development path; the other portions are optional. The other portions are peripheral functions that enhance the development process.

Figure 1-1. TMS320C55x Software Development Flow



1.2 Tools Descriptions

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

- The **C/C++ compiler** accepts C/C++ source code and produces TMS320C55x machine code object modules. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:
  - The shell program enables you to compile, assemble, and link source modules in one step.
  - The optimizer modifies code to improve the efficiency of C/C++ programs.
  - The interlist utility interlists C/C++ source statements with assembly language output to correlate code produced by the compiler with your source code.

  See the *TMS320C55x Optimizing C/C++ Compiler User's Guide* for more information.
- The **assembler** translates assembly language source files into machine language object modules. The TMS320C55x assembler accepts C54x and C55x mnemonic assembly source files, or C55x algebraic assembly source files. Source files can contain instructions, assembler directives, and macro directives. You can use assembler directives to control various aspects of the assembly process, such as the source listing format, data alignment, and section content. See Chapter 3 through Chapter 5. See the *TMS320C55x DSP Algebraic Instruction Set Reference Guide*, the *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*, for detailed information on the assembly language instruction set.
- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable object modules (created by the assembler) as input. It also accepts archiver library members and output modules created by a previous linker run. Link directives allow you to combine object file sections, bind sections or symbols to addresses or within memory ranges, and define or redefine global symbols. See Chapter 9.
- The **archiver** allows you to collect a group of files into a single archive file, called a library. You can also use the archiver to collect a group of object files into an object library. 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. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members. See Section 8.1.
- The **library information archiver** allows you to create an index library of several object file library variants, which is useful when several variants of a library with different options are available. Rather than refer to a specific library, you can link against the index library, and the linker will choose the best match from the indexed libraries. See Section 8.5.
- You can use the **library-build utility** to build your own customized run-time-support library. See the *TMS320C55x Optimizing C/C++ Compiler User's Guide* for more information.
- The **hex conversion utility** converts an object file into TI-Tagged, ASCII-Hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer. See Chapter 13.
- The **absolute lister** uses linked object files to create .abs files. These files can be assembled to produce a listing of the absolute addresses of object code. See Chapter 10.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definition, and their references in the linked source files. See Chapter 11.
- The main product of this development process is a executable object file that can be executed in a **TMS320C55x** device. You can use one of several debugging tools to refine and correct your code. Available products include:
  - An instruction-level and clock-accurate software simulator
  - An extended development system (XDS510E™ emulator

In addition, the following utilities are provided:

- The **object file display utility** prints the contents of object files, executable files, and archive libraries in either human readable or XML formats. See Section 12.1.
- The **disassembler** decodes object modules to show the assembly instructions that it represents. See Section 12.2.
- The **name utility** prints a list of linknames of objects and functions defined or referenced in a object or an executable file. See Section 12.3.
- The **strip utility** removes symbol table and debugging information from object and executable files. See Section 12.4.

Introduction to Object Modules

The assembler creates object modules from assembly code, and the linker creates executable object files from object modules. These executable object files can be executed by a TMS320C55x device.

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

This chapter focuses on the concept and use of sections in assembly language programs.

Topic **Page**

2.1 Sections

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

.text section	contains executable code [(1)]
.data section	usually contains initialized data
.bss section	usually reserves space for uninitialized variables

(1) Some targets allow non-text in .text sections.

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:

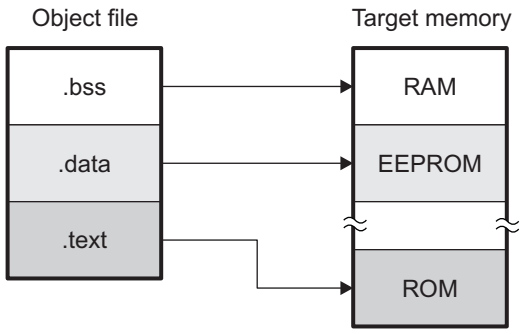
Initialized sections	contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.
Uninitialized sections	reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created 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 system's memory map; this function is called *allocation*. Because most systems contain several types of memory, using sections can help you use target memory more efficiently. All sections are independently relocatable; you can place any section into any allocated block of target memory. For example, you can define a section that contains an initialization routine and then allocate the routine into a portion of the memory map that contains ROM.

Figure 2-1 shows the relationship between sections in an object file and a hypothetical target memory.

Figure 2-1. Partitioning Memory Into Logical Blocks



2.2 How the Assembler Handles Sections

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

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

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

You can create subsections of any section to give you tighter control of the memory map. Subsections are created using the .sect and .usect directives. Subsections are identified with the base section name and a subsection name separated by a colon; see Section 2.2.4.

Default Sections Directive

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

2.2.1 Uninitialized Sections

Uninitialized sections reserve space in TMS320C55x memory; they are usually allocated into RAM. These sections have no actual contents in the object file; they simply reserve memory. A program can use this space at run time for creating and storing variables.

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

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

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

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

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 are reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the .global directive).
size in words	is an absolute expression. The .bss directive reserves *size in words* words in the .bss section. The .usect directive reserves *size in words* words in section name. For either directive you must specify a size; there is no default value.
blocking flag	is an optional parameter. If you specify a value greater than 0 for this parameter, the assembler allocates *size in words* contiguously. This means the allocated space does not cross a page boundary unless its size is greater than a page, in which case the objects starts a page boundary.
alignment flag	is an optional parameter. It causes the assembler to allocate size in words on word boundaries.
section name	tells the assembler which named section to reserve space in. See Section 2.2.3.

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

The .usect directive can also be used to create uninitialized subsections. See Section 2.2.4, for more information on creating subsections.

2.2.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in TMS320C55x memory when the program is loaded. Each initialized section is independently relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section-relative references.

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

.text

.data

.sect "*section name*"

The assembler adds code or data to one section at a time. The section the assembler is currently filling is the current section. The .text, .data, and .sect directives change the current section. When the assembler encounters one of these directives, it stops assembling into the current section (acting as an implied end of current section command). The assembler sets the designated section as the current section and assembles subsequent code into the designated section until it encounters another .text, .data, or .sect directive.

If one of these directives sets the current section to a section that already has code or data in it, the assembler resumes adding to the end of that section. The assembler generates only one contiguous section for each given section name. This section is formed by concatenating allof the code or data which was placed in that section.

Initialized subsections are created with the .sect directive. The .sect directive can also be used to create initialized subsections. See Section 2.2.4, for more information on creating subsections.

2.2.3 Named Sections

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

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

Two directives let you create named sections:

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

The syntaxes for these directives are:

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

The *section name* parameter is the name of the section. You can create up to 32 767 separate named sections. For the .usect and .sect directives, a section name can refer to a subsection; see Section 2.2.4 for details.

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

2.2.4 Subsections

Subsections are smaller sections within larger sections. Like sections, subsections can be manipulated by the linker. Placing each function and object in a uniquely-named subsection allows finer-grained memory placement, and also allows the linker finer-grained unused-function elimination. You can create subsections by using the .sect or .usect directive. The syntaxes for a subsection name are:

symbol	**.usect** "*section name*:*subsection name*"**,***size in bytes*[**,***blocking flag*[**,***alignment*]]
	.sect "*section name*:*subsection name*"

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

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

Using the linker's SECTIONS directive, you can allocate .text:_func separately, or with all the .text sections. See Section 9.6.4.1 for an example using subsections.

You can create two types of subsections:

- Initialized subsections are created using the .sect directive. See Section 2.2.2.
- Uninitialized subsections are created using the .usect directive. See Section 2.2.1.

Subsections are allocated in the same manner as sections. See Section 9.6.4 for information on the SECTIONS directive.

2.2.5 Section Program Counters

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

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

The assembler treats each section as if it began at address 0; the linker relocates each section according to its final location in the memory map. See Section 2.4 for information on relocation.

2.2.6 Using Sections Directives

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

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

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.

See Section 3.12 for more information on interpreting the fields in a source listing.

Figure 2-2. Using Sections Directives Example

```
 2                      ************************************************
 3                      **  Assemble an initialized table into .data. **
 4                      ************************************************
 5  000000                        .data
 6  000000 0011         coeff     .word     011h,022h,033h
    000001 0022
    000002 0033
 7                      ************************************************
 8                      **   Reserve space in .bss for a variable.    **
 9                      ************************************************
10  000000                        .bss      buffer,10
11                      ************************************************
12                      **              Still in .data.               **
13                      ************************************************
14  000003 0123         ptr       .word     0123h
15                      ************************************************
16                      **    Assemble code into the .text section.   **
17                      ************************************************
18  000000                        .text
19  000000 A01E         add:      MOV       0Fh,AC0
20  000002 4210         aloop:    SUB       #1,AC0
21  000004 0450                   BCC       aloop,AC0>=#0
    000006 FB
22                      ************************************************
23                      **    Another initialized table into .data.   **
24                      ************************************************
25  000004                        .data
26  000004 00AA         ivals     .word     0AAh, 0BBh, 0CCh
    000005 00BB
    000006 00CC
27                      ************************************************
28                      ** Define another section for more variables. **
29                      ************************************************
30  000000              var2      .usect    "newvars", 1
31  000001              inbuf     .usect    "newvars", 7
32                      ************************************************
33                      **       Assemble more code into .text.       **
34                      ************************************************
35  000007                        .text
36  000007 A114         mpy:      MOV       0Ah,AC1
37  000009 2272         mloop:    MOV       T3,HI(AC2)
38  00000b 1E0A                   MPYK      #10,AC2,AC1
    00000d 90
39  00000e 0471                   BCC       mloop,!overflow(AC1)
    000010 F8
40                      ************************************************
41                      **  Define a named section for int. vectors.  **
42                      ************************************************
43  000000                        .sect     "vectors"
44  000000 0011                   .word     011h, 033h
45  000001 0033

    Field 1 Field 2  Field 3                Field 4
```

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

.text	contains 17 bytes of object code.
.data	contains seven words of initialized data.
vectors	is a named section created with the .sect directive; it contains two words of object code.
.bss	reserves 10 words in memory.
newvars	is a named section created with the .usect directive; it contains eight words in memory.

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

Figure 2-3. Object Code Generated by the File in Figure 2-2

Line numbers	Object code	Section
19 20 21 21 36 37 38 38 39 39	A01E 4210 0450 FB A114 5272 1E0A 90 0471 F8	.text
6 6 6 14 26 26 26	0011 0022 0033 0123 00aa 00bb 00cc	.data
44 45	0011 0033	vectors
10	No data - 10 words reserved	.bss
30 31	No data - 8 words reserved	newvars

2.3 How the Linker Handles Sections

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

Two linker directives support these functions:

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

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

It is not always necessary to use linker directives. If you do not use them, the linker uses the target processor's default allocation algorithm described in Section 9.8. When you *do* use linker directives, you must specify them in a linker command file.

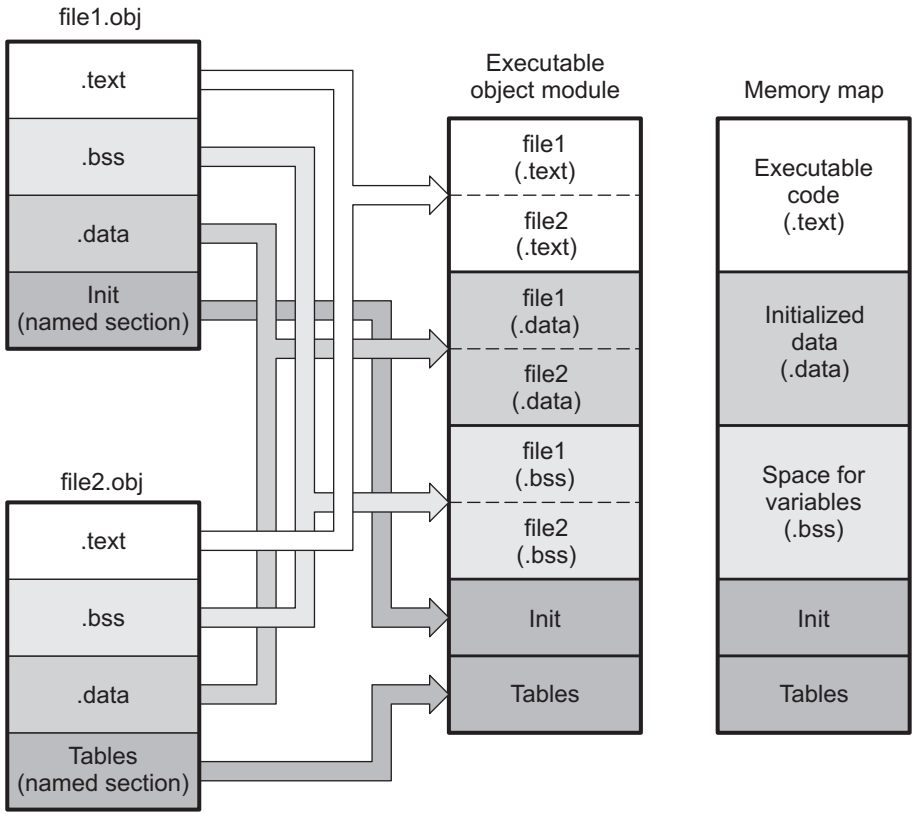
Refer to the following sections for more information about linker command files and linker directives:

- Section 9.6, *Linker Command Files*
- Section 9.6.3, *The MEMORY Directive*
- Section 9.6.4, *The SECTIONS Directive*
- Section 9.8, *Default Allocation Algorithm*

2.3.1 Default Memory Allocation

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

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



In Figure 2-4, file1.obj and file2.obj have been assembled to be used as linker input. Each contains the .text, .data, and .bss default sections; in addition, each contains a named section. The executable object module shows the combined sections. The linker combines the .text section from file1.obj and the .text section from file2.obj to form one .text section, then combines the two .data sections and the two .bss sections, and finally places the named sections at the end. The memory map shows how the sections are put into memory.

By default, the linker begins at 0h and places the sections one after the other in the following order: .text, .const, .data, .bss, .cinit, and then any named sections in the order they are encountered in the input files.

The C/C++ compiler uses the .const section to store string constants, and variables or arrays that are declared as *const*. The C/C++ compiler produces tables of data for autoinitializing global variables; these variables are stored in a named section called .cinit (see Example 9-8). For more information on the .const and .cinit sections, see the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.

2.3.2 Placing Sections in the Memory Map

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

For further explanation of section placement within the memory map, see the discussions in Section 9.6.3 and Section 9.6.4.

2.4 Relocation

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

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

The linker uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The linker then uses these entries to patch the references after the symbols are relocated. Example 2-1 contains a code segment for a TMS320C55x device that generates relocation entries.

Example 2-1. Code That Generates Relocation Entries

```
1                .ref   X
2                .ref   Z
3 000000         .text
4 000000 4A04    B   Y
5 000002 6A00    B   Z  ;Generates relocation entry
  000004 0000!
6 000006 7600    MOV #X,AC0 ;Generates relocation entry
  000008 0008!
7 00000a 9400  Y: reset
```

In Example 2-1, symbol X is relocatable since it is defined in another module. Symbol Y is relative to the PC and relocation is not necessary. Symbol Z is PC-relative and needs relocation because it is in a different file. When the code is assembled, X and Z have a value of 0 (the assembler assumes all undefined external symbols have values of 0). The assembler generates a relocation entry for X and Z. The references to X and Z are external references (indicated by the **!** character in the listing).

Each section in an object module 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 --relocatable option (see Section 9.4.2.2).

2.5 Relocation Issues

The linker may warn you about certain relocation issues.

In an assembly program, if an instruction with a PC-relative field contains a reference to a symbol, label, or address, the relative displacement is expected to fit in the instruction's field. If the displacement does not fit into the field (because the referenced item's location is too far away), the linker issues an error. For example, the linker will issue an error message when an instruction with an 8-bit, unsigned, PC-relative field references a symbol located 256 or more bytes away from the instruction.

Similarly, if an instruction with an absolute address field contains a reference to a symbol, label, or address, the referenced item is expected to be located at an address that will fit in the instruction's field. For example, if a function is linked at 0x10000, its address cannot be encoded into a 16-bit instruction field.

In both cases, the linker truncates the high bits of the value.

To deal with these issues, examine your link map and linker command file. You may be able to rearrange output sections to put referenced symbols closer to the referencing instruction.

Alternatively, consider using a different assembly instruction with a wider field. Or, if you only need the lower bits of a symbol, use a mask expression to zero out the upper bits.

2.6 Run-Time Relocation

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

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

The load address determines where a loader places the raw data for the section. Any references to the section (such as references to labels in it) refer to its run address. The application must copy the section from its load address to its run address before the first reference of the symbol is encountered at run time; this does *not* happen automatically simply because you specify a separate run address. For an example that illustrates how to move a block of code at run time, see Example 9-10.

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

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

For a complete description of run-time relocation, see Section 9.6.5.

2.7 Loading a Program

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

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. Common situations are described below:

- Code Composer Studio can load an executable object module onto hardware. The Code Composer Studio loader reads the executable file and copies the program into target memory.
- You can use the hex conversion utility (hex55, which is shipped as part of the assembly language package) to convert the executable 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 an Object File

An object file contains a symbol table that stores information about symbols in the program. The linker uses this table when it performs relocation.

2.8.1 External Symbols

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

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

The following code segment illustrates these definitions.

```
     .def   x            ; DEF of x
     .ref   y            ; REF of y
x:   ADD    #86,AC0,AC0  ; Define x
     B      y            ; Reference y
```

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

The assembler places both x and y in the object file's symbol table. When the file is linked with other object files, the entry for x resolves references to x 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 defined by one of the directives in Section 2.8.1). The assembler also creates special symbols that point to the beginning of each section; the linker uses these symbols to relocate references to other symbols.

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

Assembler Description

The TMS320C55x assembler translates assembly language source files into machine language object files. These files are in object modules, which are discussed in Chapter 2. Source files can contain the following assembly language elements:

Assembler directives	described in Chapter 4
Macro directives	described in Chapter 5
Assembly language instructions	described in the *TMS320C55x CPU Mnemonic Instruction Set Reference Guide* and the *TMS320C55x CPU Algebraic Instruction Set Reference Guide*.

Topic **Page**

3.1 Assembler Overview

TMS320C55x has two assemblers:

- The mnemonic assembler accepts C54x™ mnemonic and C55x™ mnemonic assembly source.
- The algebraic assembler accepts only C55x algebraic assembly source.

Each assembler does the following:

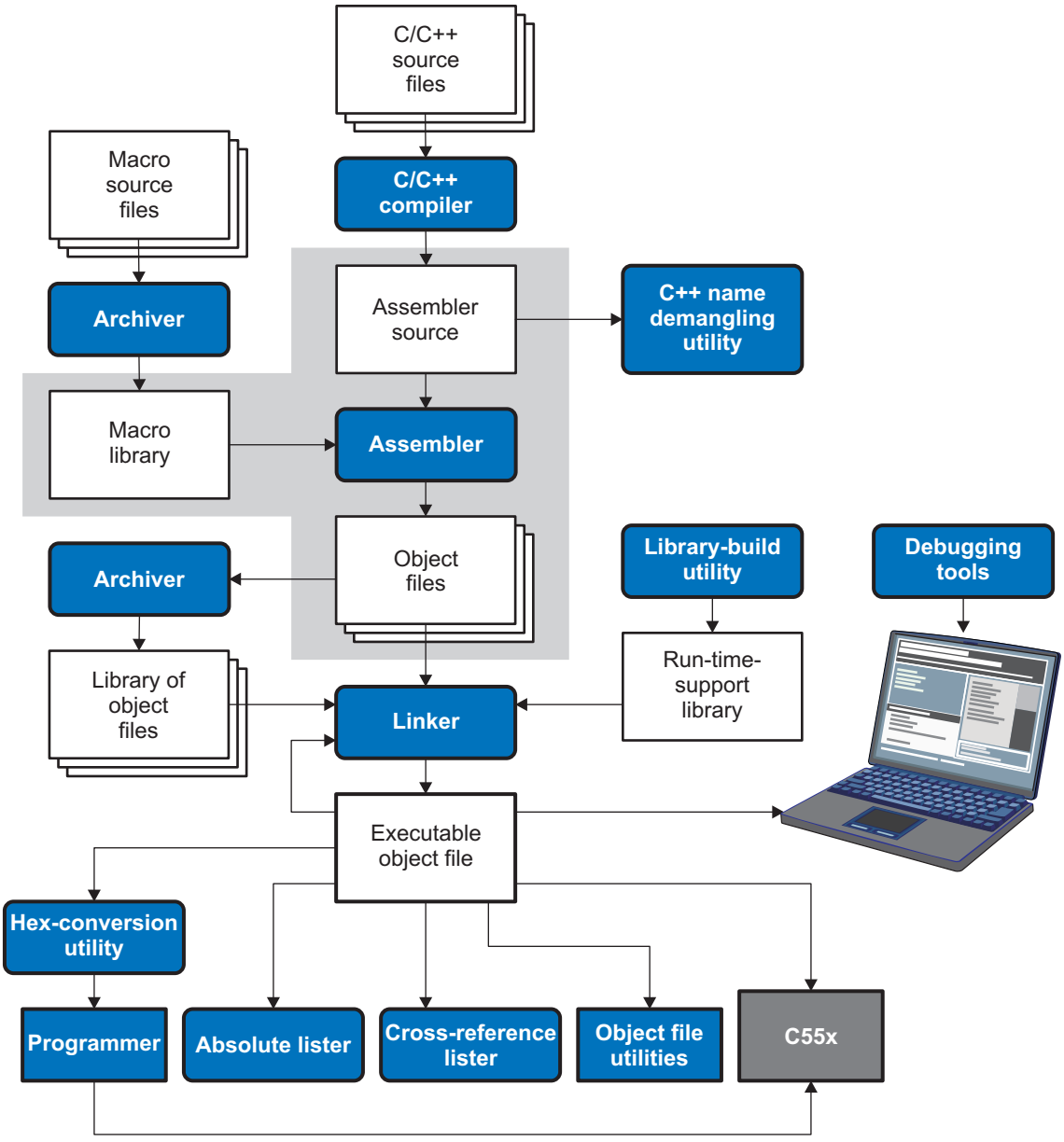
- Processes the source statements in a text file to produce a relocatable object file
- Produces a source listing (if requested) and provides you with control over this listing
- Allows you to divide your code into sections and maintain a section program counter (SPC) for each section of object code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

The mnemonic assembler generates error and warning messages for C54x instructions that are not supported. Some C54x instructions do not map directly to a single C55x instruction. The mnemonic assembler will translate these instructions into an appropriate series of C55x instructions. The listing file generated by the assembler (with the --asm_listing option) shows the translations that have occurred. See Chapter 6 for more information on running C54x code on C55x.

3.2 The Assembler's Role in the Software Development Flow

Figure 3-1 illustrates the assembler's role in the software development flow. The shaded portion highlights the most common assembler development path. The assembler accepts assembly language source files as input, both those you create and those created by the TMS320C55x C/C++ compiler.

Figure 3-1. The Assembler in the TMS320C55x Software Development Flow



3.3 Invoking the Assembler

To invoke the assembler, enter the following:

cl55 *input file* [*options*]

cl55 is the command that invokes the assembler through the compiler. The compiler considers any file with an .asm extension to be an assembly file and calls the assembler.

input file names the assembly language source file.

options identify the assembler options that you want to use. Options are case sensitive and can appear anywhere on the command line following the command. Precede each option with one or two hyphens as shown.

The valid assembler options are listed in Table 3-1.

Table 3-1. TMS320C55x Assembler Options

Option	Alias	Description
--absolute_listing	**-aa**	Creates an absolute listing. When you use --absolute_listing, the assembler does not produce an object file. The --absolute_listing option is used in conjunction with the absolute lister.
-ar=*num*		Suppresses the assembler remark identified by *num*. A remark is an informational assembler message that is less severe than a warning. If you do not specify a value for #, all remarks are suppressed. For a description of assembler remarks, see Section 7.7.
--asm_define=*name*[=*def*]	**-ad**	Sets the *name* symbol. This is equivalent to defining *name* with a .set directive in the case of a numeric value or with an .asg directive otherwise. If *value* is omitted, the symbol is set to 1. See Section 3.9.4.
--asm_dependency	**-apd**	Performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a .ppa extension.
--asm_includes	**-api**	Performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of files included with the .include directive. The list is written to a file with the same name as the source file but with a .ppa extension.
--asm_listing	**-al**	Produces a listing file with the same name as the input file with a .lst extension.
--asm_source=**algebraic**	**-mg**	Causes the assembler to accept algebraic assembly files. You must use the −asm_source option to assemble algebraic assembly input files. Algebraic and mnemonic source code cannot be mixed in a single source file.
--asm_undefine=*name*	**-au**	Undefines the predefined constant *name*, which overrides any --asm_define options for the specified constant.
-ata	**-ma**	(ARMS mode) Informs the assembler that the ARMS status bit will be enabled during the execution of this source file. By default, the assembler assumes that the bit is disabled.
-atb		Causes the assembler to treat parallel bus conflict errors as warnings.
-atc	**-mc**	(CPL mode) Tells the assembler to assume that the CPL status bit is initially set during the execution of this source file. This causes the assembler to enforce the use of SP-relative addressing syntax. By default, the assembler assumes that the bit is disabled.
-ath	**-mh**	Causes the assembler to generate faster code rather than smaller code when porting your C54x files. By default, the assembler tries to encode for small code size.
-atl	**-ml**	(C54x compatibility mode) Tells the assembler to assume the C54CM status bit will be enabled during the execution of this source file. By default, the assembler assumes that the bit is disabled.
-atn	**-mn**	Causes the assembler to remove NOPs located in the delay slots of C54x delayed branch/call instructions. For more information, see Section 7.2.4.
-atp		Causes the assembler to generate an assembly instruction profile file with an extension of .prf. The file contents are usage counts for each kind of instruction used in the assembly code.
-ats	**-ms**	(Mnemonic assembly only) Loosens the requirement that a literal shift count operand begin with a # character. This provides compatibility with early versions of the mnemonic assembler. When this option is used and the # is omitted, a warning is issued advising you to change to the new syntax.

Table 3-1. TMS320C55x Assembler Options (continued)

Option	Alias	Description
-att	**-mt**	Tells the assembler to assume that the SST status bit is zero during the execution of this source file. By default, the assembler assumes that the bit is enabled.
-atv	**-mv**	Tells the assembler to assume that all goto/calls are to be encoded as 24-bit offset. By default, the assembler tries to resolve all variable-length instructions to their smallest size.
-atw	**-mw**	(Algebraic assembly only) Suppresses all assembler warning messages.
--cmd_file=*filename*	**-@**	Appends the contents of a file to the command line. You can use this option to avoid limitations on command line length imposed by the host operating system. Use an asterisk or a semicolon (* or ;) at the beginning of a line in the command file to include comments. Comments that begin in any other column must begin with a semicolon. Within the command file, filenames or option parameters containing embedded spaces or hyphens must be surrounded with quotation marks. For example: "this-file.asm"
--copy_file=*filename*	**-ahc**	Copies the specified file for the assembly module. The file is inserted before source file statements. The copied file appears in the assembly listing files.
--cross_reference	**-ax**	Produces a cross-reference table and appends it to the end of the listing file; it also adds cross-reference information to the object file for use by the cross-reference utility. If you do not request a listing file but use the --cross_reference option, the assembler creates a listing file automatically, naming it with the same name as the input file with a .lst extension.
--include_file=*filename*	**-ahi**	Includes 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.
--include_path=*pathname*	**-I**	Specifies a directory where the assembler can find files named by the .copy, .include, or .mlib directives. There is no limit to the number of directories you can specify in this manner; each pathname must be preceded by the --include_path option. See Section 3.5.1.
--output_all_syms	**-as**	Puts all defined symbols in the object file's symbol table. The assembler usually puts only global symbols into the symbol table. When you use --output_all_syms, symbols defined as labels or as assembly-time constants are also placed in the table.
--purecirc		(Mnemonic assembly only) Asserts to the assembler that the C54x file uses C54x circular addressing (does not use the C55x linear/circular mode bits). For more information, see Section 7.2.3.
--quiet	**-q**	Suppresses the banner and progress information (assembler runs in quiet mode).
--symdebug:dwarf	**-g**	Enables assembler source debugging in the C source debugger. Line information is output to the object module for every line of source in the assembly language source file. You cannot use the --symdebug:dwarf option on assembly code that contains .line directives. See Section 3.13.
--syms_ignore_case	**-ac**	Makes case insignificant in the assembly language files. For example, --syms_ignore_case makes the symbols ABC and abc equivalent. *If you do not use this option, case is significant* (default). Case significance is enforced primarily with symbol names, not with mnemonics and register names.

3.4 C55x Assembler Features

The sections that follow provide important information on features specific to the C55x assembler:

- Byte/word addressing (Section 3.4.1)
- Parallel instruction rules (Section 3.4.2)
- Variable-length instructions (Section 3.4.3)
- Memory modes (Section 3.4.4)
- Warning on use of MMR addresses (Section 3.4.5)

3.4.1 Byte/Word Addressing

C55x memory is 8-bit byte-addressable for code and 16-bit word-addressable for data. The assembler and linker keep track of the addresses, relative offsets, and sizes of the bits in units that are appropriate for the given section: words for data sections, and bytes for code sections.

Offsets in .struct and .union Constructs

NOTE: Offsets of fields defined in .struct or .union constructs are always counted in words, regardless of the current section. The assembler assumes that a .struct or .union is always used in a data context.

3.4.1.1 Definition of Code Sections

The assembler identifies a section as a code section if one of the following is true:

- The section is introduced with a .text directive.
- The section has at least one instruction assembled into it.

If a section is not established with a .text, .data., or .sect directive, the assembler assumes that it is a .text (code) section. Because the section type determines the assembler's offset and size computations, it is important to clearly define your current working section as code or data before assembling objects into the section.

3.4.1.2 Assembly Programs and Native Units

The assembler and the linker assume that your code is written using word addresses and offsets in the context of data segments, and byte addresses and offsets in the context of code segments:

- If an address is to be sent via a program address bus (e.g., an address used as the target of a call or a branch), the processor expects a full 24-bit address. A constant used in this context should be expressed in bytes.
- If an address is to be sent via a data address bus (e.g., an address denotes a location in memory to be read or written), the processor expects a 23-bit word address. A constant used in this context should be expressed in words.
- The PC-value column of the assembly listing file is counted in units that are appropriate for the section being listed. For code sections, the PC is counted in bytes; for data sections, it is counted in words.

  For example:

```
1 000000            .text   ; PC is counted in BYTES
2 000000 2298       MOV AR1,AR0
3 000002 4010       ADD #1,AC0
4
5 000000            .data   ; PC is counted in WORDS
6 000000 0004       .word 4,5,6,7
  000001 0005               ; PC is 1 word
  000002 0006               ; PC is 2 words ...
  000003 0007
7 000004 0001 foo   .word 1
```

- The data definition directives that operate on characters (.byte, .ubyte, .char, .uchar, and .string) allocate one character per byte when in a code section, and one character to a word when in a data section. However, Texas Instruments highly recommends that you use data definition directives (see Table 4-2 for a complete listing) only in data sections.
- Directives that have a size parameter expressed in addressable units expect this parameter to be expressed in bytes for a code section, and in words for a data section.

  For example, the following aligns the PC to a 2-byte (16-bit) boundary in a code section, and to a 2-word (32-bit) boundary in a data section.

```
.align 2
```

Example 3-1 and Example 3-2 display data and code for C55x.

Example 3-1. C55x Data Example

```
.def Struct1, Struct2
.bss Struct1, 8   ; allocate 8 WORDS for Struct1
.bss Struct2, 6   ; allocate 6 WORDS for Struct2
.text
MOV *(#(Struct1 + 2)),T0 ; load 3rd WORD of Struct1
MOV *(#1000h),T1  ; 0x1000 is an absolute WORD
                  ; address (i.e., byte 0x2000)
```

Example 3-2. C55x Code Example

```
.text
.ref Func
CALL #(Func + 3) ;jump to address "Func plus 3 BYTES"
CALL #0x1000     ;0x1000 is an absolute BYTE address
```

3.4.1.3 Using Code as Data and Data as Code

The assembler does not support using a code address as if it were a data address (e.g., attempting to read or write data to program space) except when code has separate load and run memory placements. In those cases, code must be aligned to a word address. See Section 9.9 for more information.

Similarly, the assembler does not support using a data address as if it were a code address (e.g., executing a branch to a data label). This functionality cannot be supported because of the difference in the size of the addressable units: a code label address is a 24-bit byte address while a data label address is a 23-bit word address.

Consequently:

- *You should not mix code and data within one section*. All data (even constant data) should be placed into a section separate from code.
- Applications that attempt to read and write bits into program sections are dangerous and likely will not work.

3.4.2 Parallel Instruction Rules

The assembler performs semantic checking of parallel pairs of instructions in accordance with the rules specified in the *TMS320C55x Instruction Set Reference Guide*.

The assembler may swap two instructions in order to make parallelism legal. For example, both sets of instructions below are legal and will be encoded into identical object bits:

```
AC0 = AC1 || T0 = T1 ^ #0x3333
T0 = T1 ^ #0x3333 || AC0 = AC1
```

3.4.3 Variable-Length Instruction Size Resolution

By default, the assembler will attempt to resolve all stand-alone, variable-length instructions to their smallest possible size. For instance, the assembler will try to choose the smallest possible of the three available unconditional branch-to-address instructions:

```
goto L7
goto L16
goto P24
```

If the address used in a variable-length instruction is not known at assembly time (for example, if it is a symbol defined in another file), the assembler will choose the largest available form of the instruction. Of the three available branch instructions above, goto P24 will be picked.

Size resolution is performed on the following instruction groups:

```
goto L7, L16, P24
if (cond) goto l4, L8, L16, P24
call L16, P24
if (cond) call L16, P24
```

In some cases, you may want the assembler to keep the largest (P24) form of certain instructions. The P24 versions of certain instructions execute in fewer cycles than the smaller version of the same instructions. For example, “goto P24” uses 4 bytes and 3 cycles, while “goto L7” uses 2 bytes but 4 cycles.

Use the -atv assembler option or the .vli_off directive to keep the following instructions in their largest form:

```
goto P24
call P24
```

The -atv assembler option suppresses the size resolution of the above instructions within the entire file. The .vli_off and .vli_on directives can be used to toggle this behavior for regions of an assembly file. In the case of a conflict between the command line option and the directives, the directives take precedence.

All other variable-length instructions will continue to be resolved to their smallest possible size by the assembler, despite the -atv option or .vli_off directive.

The scope of the .vli_off and .vli_on directives is static and not subject to the control flow of the assembly program.

3.4.4 Memory Modes

The assembler supports three memory mode bits (or eight memory modes): C54x compatibility, CPL, and ARMS. The assembler accepts or rejects its input based on the mode specified; it may also produce different encodings for the same input based on the mode.

The memory modes correspond to the value of the C54CM, CPL, and ARMS status bits. The assembler cannot track the value of the status bits. You must use assembler directives and/or command line options to inform the assembler of the value of these bits. An instruction that modifies the value of the C54CM, CPL, or ARMS status bit must be immediately followed by an appropriate assembler directive. When the assembler is aware of changes to these bit values, it can provide useful error and warning messages about syntax and semantic violations of these modes.

3.4.4.1 C54x Compatibility Mode

C54x compatibility mode is necessary when a source file has been converted from C54x code. Until you modify your converted source code to be C55x-native code, use the -ml command line option when assembling the file, or use the .c54cm_on and .c54cm_off directives to specify C54x compatibility mode for regions of code. The .c54cm_on and .c54cm_off directives take no arguments. In the case of a conflict between the command line option and the directive, the directive takes precedence.

The scope of the .c54cm_on and .c54cm_off directives is static and not subject to the control flow of the assembly program. All assembly code between the .c54cm_on and .c54cm_off directives is assembled in C54x compatibility mode.

In C54x compatibility mode, AR0 is used instead of T0 (C55x index register) in memory operands. For example, *(AR5 + T0) is invalid in C54x compatibility mode; *(AR5 + AR0) should be used.

3.4.4.2 CPL Mode

CPL mode affects direct addressing. The assembler cannot track the value of the CPL status bit. Consequently, you must use the .cpl_on and .cpl_off directives to model the CPL value. Issue one of these directives immediately following any instruction that changes the value in the CPL bit. The .cpl_on directive is similar to the CPL status bit set to 1; it is equivalent to using the -mc command line option. The .cpl_off directive asserts that the CPL status bit is set to 0. The .cpl_on and .cpl_off directives take no arguments. In the case of a conflict between the command line option and the directive, the directive takes precedence.

The scope of the .cpl_on, .cpl_off directives is static and not subject to the control flow of the assembly program. All of the assembly code between the .cpl_on and .cpl_off directives is assembled in CPL mode.

In CPL mode (.cpl_on), direct memory addressing is relative to the stack pointer (SP). The dma syntax is *SP(dma), where dma can be a constant or a relocatable symbolic expression. The assembler encodes the value of dma into the output bits.

By default (.cpl_off), direct memory addressing (dma) is relative to the data page register (DP). The dma syntax is @dma, where dma can be a constant or a relocatable symbolic expression. The assembler computes the difference between dma and the value in the DP register and encodes this difference into the output bits.

The DP can be referenced in a file, but never defined in that file (it is set externally). Consequently, you must use the .dp directive to inform the assembler of the DP value before it is used. Issue this directive immediately following any instruction that changes the value in the DP register. The syntax of the directive is:

.dp *dp_value*

The *dp_value* can be a constant or a relocatable symbolic expression.

If the .dp directive is not used in a file, the assembler assumes that the value of the DP is 0. The scope of the .dp directive is static and not subject to the control flow of the program. The value set by the directive is used until the next .dp directive is encountered, or until the end of the source file is reached.

Whether CPL mode is specified or not, dma access to the MMR page and to the I/O page is processed identically by the assembler. Access to the MMR page is indicated by the mmap() qualifier in the syntax. Access to the I/O page is indicated by the readport() and writeport() qualifiers. These dma accesses are always encoded by the assembler as relative to the origin of 0.

3.4.4.3 ARMS Mode

ARMS mode affects indirect addressing and is useful in the context of controller code. The assembler cannot track the value of the ARMS status bit. Consequently, you must use the .arms_on and .arms_off directives to model the ARMS value to the assembler. Issue one of these directives immediately following any instruction that changes the value in the ARMS bit. The .arms_on directive models the ARMS status bit set to 1; it is equivalent to using the -ma option. The .arms_off directive models the ARMS status bit set to 0. The .arms_on and .arms_off directives take no arguments.

In the case of a conflict between the -ma option and the directive, the directive takes precedence.

The scope of the .arms_on and .arms_off directives is static and not subject to the control flow of the assembly program. All of the assembly code between the .arms_on and .arms_off directives is assembled in ARMS mode.

By default (.arms_off), indirect memory access modifiers targeted to the assembly code are selected.

In ARMS mode (.arms_on), short offset modifiers for indirect memory access are used. These modifiers are more efficient for code size optimization.

3.4.5 Assembler Warning On Use of MMR Address

The mnemonic assembler (cl55) issues a “Using MMR address” warning when a memory-mapped register (MMR) is used in a context where a single-memory access operand (Smem) is expected. The warning indicates that the assembler interprets the MMR usage as a DP-relative direct address operand. For the instruction to work as written, DP must be 0. For example:

```
ADD   SP, T0
```

Receives the Using MMR address warning as here:

```
"file.asm", WARNING! at line 1: [W9999] Using MMR address
```

The assembler warns that the effect of this instruction is:

```
ADD   value at address(DP + MMR address of SP), T0
```

The value of SP is accessed only if the DP is 0.

```
ADD   mmap(SP), T0
```

In a case where the DP is known to be 0 and such a reference is intentional, you can avoid the warning by using the @ prefix:

```
ADD   @SP, T0
```

This warning is not generated for C55x instructions inherited from C54x.

3.5 Naming Alternate Directories for Assembler Input

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

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

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

1. The directory that contains the current source file. The current source file is the file being assembled when the .copy, .include, or .mlib directive is encountered.
2. Any directories named with the --include_path option
3. Any directories named with the C55X_A_DIR environment variable
4. Any directories named with the C55X_C_DIR environment variable

Because of this search hierarchy, you can augment the assembler's directory search algorithm by using the --include_path option (described in Section 3.5.1) or the C55X_A_DIR environment variable (described in Section 3.5.2). The C55X_C_DIR environment variable is discussed in the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.

3.5.1 Using the --include_path Assembler Option

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

cl55 --include_path= *pathname source filename* [*other options*]

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

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

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

Assume the following paths for the copy.asm file:

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

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

Operating System	Enter
UNIX (Bourne shell)	`cl55 --include_path=/tools/files source.asm`
Windows	`cl55 --include_path=c:\tools\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 --include_path option.

3.5.2 Using the C55X_A_DIR Environment Variable

An environment variable is a system symbol that you define and assign a string to. The assembler uses the C55X_A_DIR environment variable to name alternate directories that contain copy/include files or macro libraries.

The assembler looks for the C55X_A_DIR environment variable and then reads and processes it. If the assembler does not find the C55X_A_DIR variable, it then searches for C55X_C_DIR. The processor-specific variables are useful when you are using Texas Instruments tools for different processors at the same time.

See the *TMS320C55x Optimizing C/C++ Compiler User's Guide* for details on C55X_C_DIR.

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

Operating System	Enter
UNIX (Bourne Shell)	**C55X_A_DIR="** *pathname*$_1$ **;** *pathname*$_2$ **; . . . "; export C55X_A_DIR**
Windows	**set C55X_A_DIR=** *pathname*$_1$ **;** *pathname*$_2$ **; . . .**

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

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

  ```
  set C55X_A_DIR= c:\path\one\to\tools ; c:\path\two\to\tools
  ```

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

  ```
  set C55X_A_DIR=c:\first path\to\tools;d:\second path\to\tools
  ```

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

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

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

Assume the following paths for the files:

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

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

Operating System	Enter
UNIX (Bourne shell)	`C55X_A_DIR="/dsys"; export C55X_A_DIR` `cl55 --include_path=/tools/files source.asm`
Windows	`C55X_A_DIR=c:\dsys` `cl55 --include_path=c:\tools\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 --include_path option and finds copy1.asm. Finally, the assembler searches the directory named with C55X_A_DIR and finds copy2.asm.

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

Operating System	Enter
UNIX (Bourne shell)	`unset C55X_A_DIR`
Windows	`set C55X_A_DIR=`

3.6 Source Statement Format

TMS320C55x assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. A source statement can contain four ordered fields (label, mnemonic, operand list, and comment). Source statement lines can be as long as the source file format allows. The general syntax for source statements is as follows:

Mnemonic syntax:

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

Algebraic syntax:

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

Following are examples of source statements:

Mnemonic instructions

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

Algebraic instructions

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

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- A statement containing an assembler directive must be specified entirely on one line.
- Labels are optional; if used, they must begin in column 1.
- One or more space or tab characters must separate each field.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column *must* begin with a semicolon.
- A source line can be continued onto the next line by ending the first line with a backslash (\) character.
- A mnemonic cannot begin in column 1 or it will be interpreted as a label. The assembler does **not** check to make sure you do not have a mnemonic in the label field.

The following sections describe each of the fields.

3.6.1 Label Field

Labels are optional for all assembly language instructions and for most (but not all) assembler directives. When used, a label must begin in column 1 of a source statement. A label can contain up to 32 alphanumeric characters (A-Z, a-z, 0-9, _, and $). Labels are case sensitive (except when the --syms_ignore_case option is used), and the first character cannot be a number. A label can be followed by a colon (:). The colon is not treated as part of the label name. If you do not use a label, the first character position must contain a blank, a semicolon, or an asterisk.

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

```
5 000000               .data
6 000000 00         ; Assume other code was assembled.
7       ...
8       ...
9 000040 000A  Start:  .word 0Ah,3,7
  000041 0003
  000042 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 .equ  $  ; $ provides the current value of the SPC
```

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

```
3 000043          Here:
4 000043 0003           .word 3
```

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

3.6.2 Mnemonic Instruction Fields

In mnemonic assembly, the label field is followed by the mnemonic and operand list fields. These fields are described in the next two sections.

3.6.2.1 Mnemonic Field

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

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

3.6.2.2 Operand Field

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

- Constants (see Section 3.7)
- Floating-point constants (see Section 3.7.7)
- Character strings (see Section 3.8)
- Symbols (see Section 3.9)
- Expressions (combination of constants and symbols; see Section 3.10)

You must separate operands with commas.

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, AC0
  ```

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

  For instructions that have an embedded shift count, the # prefix on the shift count operand is required. If you want the shift performed by the instruction, you must use # on the shift count.

- *** 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:   MOV *AR4,AC0
  ```

  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 the specified accumulator.

3.6.3 Algebraic Instruction Fields

In algebraic assembly, instructions are written in a form that resembles algebraic mathematical expression. The semantics of the instruction are embodied in the operators of the expression. The terms of the expression specify what operands are being acted on.

The following items describe how to use the instruction field for algebraic syntax:

- Generally, operands are not separated by commas. Some algebraic instructions consist of a mnemonic and operands. For algebraic statements of this type, commas are used to separate operands. For example, lms(Xmem, Ymem, ACx, ACy).
- 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 enclosed in parentheses. For example, AC0 = AC1 & #(1 << sym) << 5. The expression 1 << sym is used as a single operand and must therefore be delimited with parentheses.
- All register names are reserved.

3.6.4 Comment Field

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

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

3.7 Constants

The assembler supports several 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 00FFh is equal to 00FF (base 16) or 255 (base 10); it *does not* equal -1. However, when used with the .byte directive, -1 *is* equivalent to 00FFh.

In C55x algebraic assembly source code, most constants must begin with a '#'.

3.7.1 Binary Integers

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

00000000B	Constant equal to 0_{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 $0F8_{16}$

3.7.2 Octal Integers

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

10Q	Constant equal to 8_{10} or 8_{16}
010	Constant equal to 8_{10} or 8_{16} © format)
100000Q	Constant equal to 32 768_{10} or 8000_{16}
226q	Constant equal to 150_{10} or 96_{16}

3.7.3 Decimal Integers

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

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

3.7.4 Hexadecimal Integers

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

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

3.7.5 Character Constants

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

'a'	Defines the character constant *a* and is represented internally as 61_{16}
'C'	Defines the character constant *C* and is represented internally as 43_{16}
'"'	Defines the character constant ' and is represented internally as 27_{16}
''	Defines a null character and is represented internally as 00_{16}

Notice the difference between character *constants* and character *strings* (Section 3.8 discusses character strings). A character constant represents a single integer value; a string is a sequence of characters.

3.7.6 Assembly-Time Constants

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

```
shift3   .set 3
         MOV AR1, #shift3
```

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

```
myReg   .set AR1
        MOV  myReg, #3
```

3.7.7 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
```

The .double directive converts a floating-point constant into a floating-point value in IEEE double-precision 64-bit format. The .float directive converts a floating-point constant into a floating-point value in IEEE single-precision 32-bit format. See Initialize Double-Precision Floating-Point Value and Initialize Single-Precision Floating-Point Value for more information.

3.8 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

3.9 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of alphanumeric characters, the dollar sign, and underscores (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 --syms_ignore_case assembler option (see Section 3.3). A symbol is valid only during the assembly in which it is defined, unless you use the .global directive or the .def directive to declare it as an external symbol (see Identify Global Symbols).

3.9.1 Labels

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

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

```
          .global  label1
label2    nop
          ADD @label1,AC1,AC1
          B label2
```

3.9.2 Local Labels

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

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

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

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

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

Example 3-3. Local Labels of the Form $n

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

```
Label1:   MOV ADDRA,AC0      ; Load Address A to AC0.
          SUB ADDRB,AC0,AC0  ; Subtract Address B.
          BCC $1,AC0 < #0    ; If < 0, branch to $1
          MOV ADDRB,AC0 ; otherwise, load ADDRB to AC0
          B $2           ; and branch to $2.

$1        MOV ADDRA,AC0      ; $1: load ADDRA to AC0.

$2        ADD ADDRC,AC0,AC0  ; $2: add ADDRC.
          .newblock     ; Undefine $1 so it can be used again.
          BCC $1,AC0 < #0    ; If less than zero, branch to $1.
          MOV AC0,ADDRC      ; Store AC0 low in ADDRC.

$1        NOP
```

The following code uses a local label illegally:

```
Label1:   MOV ADDRA,AC0
          SUB ADDRB,AC0,AC0
          BCC $1,AC0 < #0
          MOV ADDRB,AC0
          B $2
$1        MOV ADDRA,AC0
$2        ADD ADDRC,AC0,AC0
          BCC $1,AC0 < #0
          MOV AC0,ADDRC
$1        NOP           ; Wrong: $1 is multiply defined.
```

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

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

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

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

Example 3-4. Local Labels of the Form name?

```
; 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?

; 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
          .data
          mylab? .int 0
          .text
          nop
          nop
          B mylab?

; .newblock directive, allowing sixth definition of 'mylab'
          .newblock
          .data
mylab?    .int 0
          .text
          nop
          nop
          B mylab?
```

For more information about using labels in macros see Section 5.6.

3.9.3 Symbolic Constants

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

```
K      .set  1024            ; constant definitions
maxbuf .set  2*K
value  .set  0
delta  .set  1

item   .struct               ; item structure definition
       .int  value
       .int  delta
i_len  .endstruct            ; i_len=length of .struct (2)

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

The assembler also has several predefined symbolic constants; these are discussed in Section 3.9.5.

3.9.4 Defining Symbolic Constants (--asm_define Option)

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

cl55 --asm_define=*name*[**=***value*]

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

- For Windows, use **--asm_define=** *name* **="\"** *value* **\""**. For example, --asm_define=car="\"sedan\""
- For UNIX, use **--asm_define=** *name* **='"** *value* **"'**. For example, --asm_define=car='"sedan"'
- For Code Composer, enter the definition in a file and include that file with the --cmd_file (or -@) option.

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

```
cl55 --asm_define=SYM1=1 --asm_define=SYM2=2 --asm_define=SYM3=3 --asm_define=SYM4=4 value.asm
```

Since you have assigned values to SYM1, SYM2, SYM3, and SYM4, you can use them in source code. Example 3-5 shows how the value.asm file uses these symbols without defining them explicitly.

Within assembler source, you can test the symbol defined with the --asm_define option 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*

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.

Example 3-5. Using Symbolic Constants Defined on Command Line

```
IF_4:  .if       SYM4 = SYM2 * SYM2
       .byte     SYM4          ; Equal values
       .else
       .byte     SYM2 * SYM2   ; Unequal values
       .endif

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

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

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

3.9.5 Predefined Symbolic Constants

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

- **$**, the dollar-sign character, represents the current value of the section program counter (SPC). $ is a relocatable symbol.
- The symbols **.ALGEBRAIC** and **.MNEMONIC** are true when the corresponding assembly format is in effect, and false otherwise.
- The symbols, __**SMALL_MODEL**__, __**LARGE_MODEL**__, and __**HUGE_MODEL**__ are predefined and set to the value 1 if the corresponding memory model specified for this compilation, and set to 0 otherwise. You can use this symbol to write memory-model independent code such as:

  ```
  .if   __LARGE_MODEL__|__HUGE_MODEL__
  AMOV  #addr, XAR2 ; load 23-bit address
  .else
  AMOV  #addr, AR2  ; load 16-bit address
  .endif
  ```

  For more information on the memory models, see the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.
- The symbol __**TI_ASSEMBLER_VERSION**__ is set to an integer indicating the assembler version number. Version X.Y.Z is represented by XXXYYYZZZ where each portion, X, Y and Z, is expanded to three digits and concatenated together. For example, 3.2.1 is represented as 3002001. You can use this symbol to write code that will be assembled conditionally according to the assembler version:

  ```
  .if    __TI_ASSEMBLER_VERSION__ == 3002001
  .word  0x110
  .endif
  .if    __TI_ASSEMBLER_VERSION__ == 4003003
  .word  0x120
  .endif
  ```
- The assembler sets up predefined symbols for you to refer to all of the **memory-mapped registers**.

3.9.6 Substitution Symbols

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

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

```
.asg  "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

      MOV ADDRA,AC0
      ADD ADDRB,AC0,AC0
      MOV AC0,ADDRB
     .endm

; add2 invocation
      add2 LOC1, LOC2

; the macro will be expanded as follows:
      MOV LOC1,AC0
      ADD LOC2,AC0,AC0
      MOV AC0,LOC2
```

See Chapter 5 for more information about macros.

3.10 Expressions

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

Parentheses	Expressions 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	Operators, listed in Table 3-2, are divided into nine precedence groups. When parentheses do not determine the order of expression evaluation, the highest precedence operation is evaluated first. 8 + 4 / 2 = 10 (4 / 2 is evaluated first)
Left-to-right evaluation	When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated from left to right, except for Group 1, which is evaluated from right to left. 8 / 4*2 = 4, but 8 / (4*2) = 1

3.10.1 Operators

Table 3-2 lists the operators that can be used in expressions, according to precedence group.

Table 3-2. Operators Used in Expressions (Precedence)

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

(1) Group 1 operators are evaluated right to left. All other operators are evaluated left to right.
(2) Unary + and - have higher precedence than the binary forms.

3.10.2 Expression Overflow and Underflow

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

3.10.3 Well-Defined Expressions

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

This is an example of a well-defined expression:

```
1000h+X
```

where X was previously defined as an absolute symbol.

3.10.4 Conditional Expressions

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

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

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

3.11 Built-in Functions

The assembler supports built-in functions for conversions and various math computations. Table 3-3 describes the built-in functions. The *expr* must be a constant value. See Table 5-1 for a description of the assembler's non-mathematical built-in functions.

Table 3-3. Built-In Mathematical Functions

Function	Description
$acos(*expr*)	Returns the arc cosine of *expr* as a floating-point value
$asin(*expr*)	Returns the arc sin of *expr* as a floating-point value
$atan(*expr*)	Returns the arc tangent of *expr* as a floating-point value
$atan2(*expr*, *y*)	Returns the arc tangent of *expr* as a floating-point value in range [-π, π]
$ceil(*expr*)	Returns the smallest integer not less than *expr*
$cos(*expr*)	Returns the cosine of *expr* as a floating-point value
$cosh(*expr*)	Returns the hyperbolic cosine of *expr* as a floating-point value
$cvf(*expr*)	Converts *expr* to a floating-point value
$cvi(*expr*)	converts *expr* to integer value
$exp(*expr*)	Returns the exponential function e^{expr}
$fabs(*expr*)	Returns the absolute value of *expr* as a floating-point value
$floor(*expr*)	Returns the largest integer not greater than *expr*
$fmod(*expr*, *y*)	Returns the remainder of *expr1* ÷ *expr2*
$int(*expr*)	Returns 1 if *expr* has an integer value; else returns 0. Returns an integer.
$ldexp(*expr*, *expr2*)	Multiplies *expr* by an integer power of 2. That is, *expr1* × 2^{expr2}
$log(*expr*)	Returns the natural logarithm of *expr*, where *expr*>0
$log10(*expr*)	Returns the base 10 logarithm of *expr*, where *expr*>0
$max(*expr1*, *expr2*)	Returns the maximum of two values
$min(*expr1*, *expr2*)	Returns the minimum of two values
$pow(*expr1*, *expr2*)	Returns *expr1*raised to the power of *expr2*
$round(*expr*)	Returns *expr* rounded to the nearest integer
$sgn(*expr*)	Returns the sign of *expr*.
$sin(*expr*)	Returns the sine of *expr*
$sinh(*expr*)	Returns the hyperbolic sine of *expr* as a floating-point value
$sqrt(*expr*)	Returns the square root of expr, expr≥0, as a floating-point value
$strtod(*str*)	Converts a character string to a double precision floating-point value. The string contains a properly-formatted C99-style floating-point constant. C99-style constants are otherwise not accepted anywhere in the tools.
$tan(*expr*)	Returns the tangent of *expr* as a floating-point value
$tanh(*expr*)	Returns the hyperbolic tangent of *expr* as a floating-point value
$trunc(*expr*)	Returns *expr* rounded toward 0

3.12 Source Listings

A source listing shows source statements and the object code they produce. To obtain a listing file, invoke the assembler with the --asm_listing option (see Section 3.3).

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

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

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

Field 1: Source Statement Number

Line number

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

Include file letter

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

Nesting level number

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

Field 2: Section Program Counter

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

Field 3: Object Code

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

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

Field 4: Source Statement Field

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

Figure 3-2 and Figure 3-3 show assembler listings with each of the four fields identified.

Figure 3-2. Mnemonic Assembly Listing

```
       1                            .global RSET, 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                            BSET #9,ST1_55  ;disable overflow
       9                            MOV #0,DP       ;set dp
      10                            MOV #55,AC0     ;set AC0
      11                            BCLR #11,ST1_55 ;enable ints
      12                            .endm
      13                   ****************************************
      14                   *       Reset and interrupt vectors       *
      15                   ****************************************
      16   000000                   .sect "rset"
      17   000000  6A00     RSET:   B init
           000002  0010+
      18   000004  6A00     INT0:   B ISR0
           000006  0000!
      19   000008  6A00     INT1:   B ISR1
           00000a  0000!
      20   00000c  6A00     INT2:   B ISR2
           00000e  0000!
      21
      22                   *
      23   000000                   .sect "ints"
      24   000000  6A00     TINT    B time
           000002  0000!
      25   000004  6A00     RINT    B rcv
           000006  0000!
      26   000008  6A00     XINT    B xmt
           00000a  0000!
      27   00000c  6A00     USER    B proc
           00000e  0000!
      28                   ****************************************
      29                   *         Initialize processor.          *
      30                   ****************************************
      31   000010          init:   initmac
1                          * initialize macro
1          000010  4693            BSET #9,ST1_55
           000012  7800            MOV #0,DP
           000014  0000
1          000016  7600            MOV #55,AC0
           000018  3708
1          00001a  46B2            BCLR #11,ST1_55
```

Field 1 Field 2 Field 3 Field 4

Figure 3-3. Algebraic Assembly Listing

```
       1                                .global RSET, 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                             bit(ST1, #ST1_SATD) = #1 ;disable oflow
       9                             DP = #((01FFH & 0) << 7) ;set dp
      10                             AC0 = #55              ;set AC0
      11                             bit(ST1, #ST1_INTM) = #0 ;enable ints
      12                             .endm
      13                   ****************************************
      14                   *       Reset and interrupt vectors       *
      15                   ****************************************
      16   000000                    .sect "rset"
      17   000000  6A00      RSET:   goto  #(init)
           000002  0010+
      18   000004  6A00      INT0:   goto  #(ISR0)
           000006  0000!
      19   000008  6A00      INT1:   goto  #(ISR1)
           00000a  0000!
      20   00000c  6A00      INT2:   goto  #(ISR2)
           00000e  0000!
      21
      22                   *
      23   000000                    .sect "ints"
      24   000000  6A00      TINT    goto  #(time)
           000002  0000!
      25   000004  6A00      RINT    goto  #(rcv)
           000006  0000!
      26   000008  6A00      XINT    goto  #(xmt)
           00000a  0000!
      27   00000c  6A00      USER    goto  #(proc)
           00000e  0000!
      28                   ****************************************
      29                   *         Initialize processor.        *
      30                   ****************************************
      31   000010          init:   initmac
1                          * initialize macro
1          000010  4693            bit(ST1, #ST1_SATD) = #1
1          000012  7800            DP = #((01FFH & 0) << 7)
           000014  0000
1          000016  7600            AC0 = #55
           000018  3708
1          00001a  46B2            bit(ST1, #ST1_INTM) = #0
```

Field 1 Field 2 Field 3 Field 4

3.13 Debugging Assembly Source

When you invoke cl55 with --symdebug:dwarf (or -g) when compiling an assembly file, the assembler provides symbolic debugging information that allows you to step through your assembly code in a debugger rather than using the Disassembly window in Code Composer Studio. This enables you to view source comments and other source-code annotations while debugging.

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

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

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

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

Example 3-6 shows the cvar.c C program that defines a variable, svar, as the structure type X. The svar variable is then referenced in the addfive.asm assembly program in Example 3-7 and 5 is added to svar's second data member.

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

```
cl55 --symdebug:dwarf cvars.c addfive.asm --run_linker --library=lnk.cmd --library=rts55.lib
     --output_file=addfive.out
```

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

Example 3-6. Viewing Assembly Variables as C Types C Program

```
typedef struct
{
   int m1;
   int m2;
} X;
X svar = { 1, 2 };
```

Example 3-7. Assembly Program for Example 3-6

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

;-------------------------------------------------------------------------
; addfive() - Add five to the second data member of _svar
;-------------------------------------------------------------------------
          .text
          .align 4
          .global addfive
addfive:  .asmfunc
           ADD #5,*abs16(#(_svar+1)) ; add 5 to svar.m2
           RET                       ; return from function
          .endasmfunc
```

3.14 Cross-Reference Listings

A cross-reference listing shows symbols and their definitions. To obtain a cross-reference listing, invoke the assembler with the --cross_reference option (see Section 3.3) or use the .option directive with the X operand (see Select Listing Options). The assembler appends the cross-reference to the end of the source listing. Example 3-8 shows the four fields contained in the cross-reference listing.

Example 3-8. An Assembler Cross-Reference Listing

```
LABEL                           VALUE     DEFN     REF
INT0                            000004+   25       5
INT1                            000008+   27       5
INT2                            00000c+   29       5
ISR0                            REF        9       25
ISR1                            REF        9       27
ISR2                            REF        9       29
RINT                            000004+   37       7
RSET                            000000+   23       5
TINT                            000000+   35       7
XINT                            000008+   39       7
init                            000010+   45       23
```

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

Value column contains an 8-digit hexadecimal number (which is the value assigned to the symbol) *or* a name that describes the symbol's attributes. A value may also be preceded by a character that describes the symbol's attributes. Table 3-4 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 3-4. Symbol Attributes

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

Assembler Directives

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

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

This chapter is divided into two parts: the first part (Section 4.1 through Section 4.12) describes the directives according to function, and the second part (Section 4.13) is an alphabetical reference.

Topic **Page**

4.1 Directives Summary

Table 4-1 through Table 4-17 summarize the assembler directives.

Besides the assembler directives documented here, the TMS320C55x software tools support the following directives:

- The assembler uses several directives for macros. Macro directives are discussed in Chapter 5; they are not discussed in this chapter.
- The C compiler uses directives for symbolic debugging. Unlike other directives, symbolic debugging directives are not used in most assembly language programs. Appendix A discusses these directives; they are not discussed in this chapter.

Labels and Comments Are Not Shown in Syntaxes

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

Table 4-1. Directives That Define Sections

Mnemonic and Syntax	Description	See
.bss *symbol, size in bytes*[**,** *alignment* [**,** *bank offset*]]	Reserves *size* bytes in the .bss (uninitialized data) section	.bss topic
.clink	Enables conditional linking for the current or specified section	.clink topic
.data	Assembles into the .data (initialized data) section	.data topic
.sect "*section name*"	Assembles into a named (initialized) section	.sect topic
.text	Assembles into the .text (executable code) section	.text topic
symbol **.usect** "*section name*"**,** *size in bytes* [*blocking*][**,** *alignment*]	Reserves *size* bytes in a named (uninitialized) section	.usect topic

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

Mnemonic and Syntax	Description	See
.byte $value_1$[**,** ... **,** $value_n$]	Initializes one or more successive bytes in the current section	.byte topic
.char $value_1$[**,** ... **,** $value_n$]	Initializes one or more successive bytes in the current section	.char topic
.cstring {$expr_1$|"$string_1$"}[**,**... **,** {$expr_n$|"$string_n$"}]	Initializes one or more text strings	.string topic
.double $value_1$[**,** ... **,** $value_n$]	Initializes one or more 64-bit, IEEE double-precision, floating-point constants	.double topic
.field *value*[**,** *size*]	Initializes a field of *size* bits (1-32) with *value*	.field topic
.float $value_1$[**,** ... **,** $value_n$]	Initializes one or more 32-bit, IEEE single-precision, floating-point constants	.float topic
.half $value_1$[**,** ... **,** $value_n$]	Initializes one or more 16-bit integers (halfword)	.half topic
.int $value_1$[**,** ... **,** $value_n$]	Initializes one or more 16-bit integers	.int topic
label: **.ivec** [*address*[, *stack mode*]]	Initializes an entry in the interrupt vector table	.ivec topic
.ldouble $value_1$[**,** ... **,** $value_n$]	Initializes one or more 64-bit, IEEE double-precision, floating-point constants	.double topic
.long $value_1$[**,** ... **,** $value_n$]	Initializes one or more 32-bit integers	.long topic
.pstring "$string_1$"[**,** ... **,** "$string_n$"]	Initializes one or more packed text strings	.pstring topic
.short $value_1$[**,** ... **,** $value_n$]	Initializes one or more 16-bit integers (halfword)	.short topic
.string {$expr_1$|"$string_1$"}[**,**... **,** {$expr_n$|"$string_n$"}]	Initializes one or more text strings	.string topic
.ubyte $value_1$[**,** ... **,** $value_n$]	Initializes one or more successive unsigned bytes in the current section	.ubyte topic

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

Mnemonic and Syntax	Description	See
.uchar *value*$_1$[, ... , *value*$_n$]	Initializes one or more successive unsigned bytes in the current section	.uchar topic
.uhalf *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers (halfword)	.uhalf topic
.uint *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers	.uint topic
.ulong *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 32-bit integers	.ulong topic
.ushort *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers (halfword)	.ushort topic
.uword *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers	.uword topic
.word *value*$_1$[, ... , *value*$_n$]	Initializes one or more 16-bit integers	.word topic
.xfloat *value*$_1$[, ... , *value*$_n$]	Initializes one or more 32-bit, IEEE single-precision, floating-point constants, but does not align on long word boundary	.xfloat topic
.xlong *value*$_1$[, ... , *value*$_n$]	Initializes one or more 32-bit integers, but does not align on long word boundary	.xlong topic

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

Mnemonic and Syntax	Description	See
.align [*size in bytes*]	Aligns the SPC on a boundary specified by *size in bytes*, which must be a power of 2; defaults to 128-byte or 128-word boundary	.align topic
.even	Equivalent to .align 2.	.even topic
.localalign	Aligns the start of a local repeat block to allow maximum localrepeat loop size	.localalign topic
.space *size*	Reserves *size* bytes in the current section; a label points to the beginning of the reserved space	.space topic

Table 4-4. Directives That Format the Output Listing

Mnemonic and Syntax	Description	See
.drlist	Enables listing of all directive lines (default)	.drlist topic
.drnolist	Suppresses listing of certain directive lines	.drnolist topic
.fclist	Allows false conditional code block listing (default)	.fclist topic
.fcnolist	Suppresses false conditional code block listing	.fcnolist topic
.length [*page length*]	Sets the page length of the source listing	.length topic
.list	Restarts the source listing	.list topic
.mlist	Allows macro listings and loop blocks (default)	.mlist topic
.mnolist	Suppresses macro listings and loop blocks	.mnolist topic
.nolist	Stops the source listing	.nolist topic
.option *option*$_1$ [, *option*$_2$, . . .]	Selects output listing options; available options are B, L, M, R, T, W, and X	.option topic
.page	Ejects a page in the source listing	.page topic
.sslist	Allows expanded substitution symbol listing	.sslist topic
.ssnolist	Suppresses expanded substitution symbol listing (default)	.ssnolist topic
.tab *size*	Sets tab to *size* characters	.tab topic
.title "*string*"	Prints a title in the listing page heading	.title topic
.width [*page width*]	Sets the page width of the source listing	.width topic

Table 4-5. Directives That Reference Other Files

Mnemonic and Syntax	Description	See
.copy ["]*filename*["]	Includes source statements from another file	.copy topic
.include ["]*filename*["]	Includes source statements from another file	.include topic
.mlib ["]*filename*["]	Specifies a macro library from which to retrieve macro definitions	.mlib topic

Table 4-6. Directives That Effect Symbol Linkage and Visibility

Mnemonic and Syntax	Description	See
.def $symbol_1$[**,** ... **,** $symbol_n$]	Identifies one or more symbols that are defined in the current module and that can be used in other modules	.def topic
.global $symbol_1$[**,** ... **,** $symbol_n$]	Identifies one or more global (external) symbols	.global topic
.ref $symbol_1$[**,** ... **,** $symbol_n$]	Identifies one or more symbols used in the current module that are defined in another module	.ref topic
.symdepend *dst symbol name*[**,** *src symbol name*]	Creates an artificial reference from a section to a symbol	.symdepend topic

Table 4-7. Directives That Enable Conditional Assembly

Mnemonic and Syntax	Description	See
.break [*well-defined expression*]	Ends .loop assembly if *well-defined expression* is true. When using the .loop construct, the .break construct is optional.	.break topic
.else	Assembles code block if the .if *well-defined expression* is false. When using the .if construct, the .else construct is optional.	.else topic
.elseif *well-defined expression*	Assembles code block if the .if *well-defined expression* is false and the .elseif condition is true. When using the .if construct, the .elseif construct is optional.	.elseif topic
.endif	Ends .if code block	.endif topic
.endloop	Ends .loop code block	.endloop topic
.if *well-defined expression*	Assembles code block if the *well-defined expression* is true	.if topic
.loop [*well-defined expression*]	Begins repeatable assembly of a code block; the loop count is determined by the *well-defined expression*.	.loop topic

Table 4-8. Directives That Define Union or Structure Types

Mnemonic and Syntax	Description	See
.cstruct	Acts like .struct, but adds padding and alignment like that which is done to C structures	.cstruct topic
.cunion	Acts like .union, but adds padding and alignment like that which is done to C unions	.cunion topic
.emember	Sets up C-like enumerated types in assembly code	Section 4.9
.endenum	Sets up C-like enumerated types in assembly code	Section 4.9
.endstruct	Ends a structure definition	.cstruct topic, .struct topic
.endunion	Ends a union definition	.cunion topic, .union topic
.enum	Sets up C-like enumerated types in assembly code	Section 4.9
.union	Begins a union definition	.union topic
.struct	Begins structure definition	.struct topic
.tag	Assigns structure attributes to a label	.cstruct topic, .struct topic .union topic

Table 4-9. Directives That Define Symbols at Assembly Time

Mnemonic and Syntax	Description	See
.asg ["]*character string*["], *substitution symbol*	Assigns a character string to *substitution symbol*	.asg topic
symbol **.equ** *value*	Equates *value* with *symbol*	.equ topic
.eval *well-defined expression* , *substitution symbol*	Performs arithmetic on a numeric *substitution symbol*	.eval topic
.label *symbol*	Defines a load-time relocatable label in a section	.label topic
.newblock	Undefines local labels	.newblock topic
symbol **.set** *value*	Equates *value* with *symbol*	.set topic
.unasg *symbol*	Turns off assignment of *symbol* as a substitution symbol	.unasg topic
.undefine *symbol*	Turns off assignment of *symbol* as a substitution symbol	.unasg topic

Table 4-10. Directives That Communicate Run-Time Environment Details

Mnemonic and Syntax	Description	See
.dp *DP_value*	Specifies the value of the DP register	.dp topic
.lock_off	Asserts the lock() modifier is not legal; resumes the default behavior	.lock_off topic
.lock_on	Identifies the beginning of a block of code that contains read-modify-write instructions	.lock_on topic
.vli_off	Identifies the beginning of a block of code in which the assembler uses the largest form of certain variable-length instructions	.vli_off topic
.vli_on	Resumes the default behavior of resolving variable-length instructions to their smallest form	.vli_on topic

Table 4-11. Directives That Relate to C55x Addressing Modes

Mnemonic and Syntax	Description	See
.arms_off	Resumes the default behavior of the assembler using indirect memory access modifiers	.arms_off topic
.arms_on	Identifies the beginning of a block of code to be assembled in ARMS mode	.arms_on topic
.c54cm_off	Resumes the default behavior of C55x code	.c54cm_off topic
.c54cm_on	Identifies the beginning of a block of C54x compatibility mode code (code that has been translated from C54x code)	.c54cm_on topic
.cpl_off	Resumes the default behavior of dma relative to DP	.cpl_off topic
.cpl_on	Identifies the beginning of a block of code to be assembled in CPL mode (dma relative to SP	.cpl_on topic

Table 4-12. Directives That Affect Porting C54x Mnemonic Assembly

Mnemonic and Syntax	Description	See
.port_for_size	Resumes the default behavior of optimizing C54x code for smaller size	.port_for_size topic
.port_for_speed	Identifies the beginning of a block of code in which the assembler optimizes ported C54x code for speed	.port_for_speed topic
.sst_off	Identifies the beginning of a block of code in which the assembler assumes the SST bit is disabled	.sst_off topic
.sst_on	Resumes the default behavior of assuming the SST bit is enabled	.sst_on topic

Table 4-13. Directives That Create or Effect Macros

Mnemonic and Syntax	Description	See
.endm	End macro definition	.endm topic
.loop[*well-defined expression*]	Begins repeatable assembly of a code block; the loop count is determined by the *well-defined expression*.	.loop topic
macname **.macro** [*parameter*$_1$][**,**... **,** *parameter*$_n$]	Define macro by *macname*	.macro topic
.mexit	Go to .endm	Section 5.2
.mlib *filename*	Identify library containing macro definitions	.mlib topic
.var	Adds a local substitution symbol to a macro's parameter list	.var topic

Table 4-14. Directives That Control Diagnostics

Mnemonic and Syntax	Description	See
.emsg *string*	Sends user-defined error messages to the output device; produces no .obj file	.emsg topic
.mmsg *string*	Sends user-defined messages to the output device	.mmsg topic
.noremark[*num*]	Identifies the beginning of a block of code in which the assembler suppresses the *num* remark	.noremark topic
.remark [*num*]	Resumes the default behavior of generating the remark(s) previously suppressed by .noremark	.remark topic
.warn_off	Identifies the beginning of a block of code in which the assembler suppresses warning messages	.warn_off topic
.warn_on	Resumes the default behavior of reporting assembler warning messages	.warn_on topic
.wmsg *string*	Sends user-defined warning messages to the output device	.wmsg topic

Table 4-15. Directives That Perform Assembly Source Debug

Mnemonic and Syntax	Description	See
.asmfunc	Identifies the beginning of a block of code that contains a function	.asmfunc topic
.endasmfunc	Identifies the end of a block of code that contains a function	.endasmfunc topic

Table 4-16. Directives That Are Used by the Absolute Lister

Mnemonic and Syntax	Description	See
.setsect	Produced by absolute lister; sets a section	Chapter 10
.setsym	Produced by the absolute lister; sets a symbol	Chapter 10

Table 4-17. Directives That Perform Miscellaneous Functions

Mnemonic and Syntax	Description	See
.cdecls [*options* **,**]"*filename*"[**,** "*filename2*"[**,** ...]	Share C headers between C and assembly code. Available in algebraic assembly only.	.cdecls topic
.end	Ends program	.end topic
.sblock["]*section name*["][**,**["]*section name*["]**,**...	Designates sections for blocking. Only initialized sections can be specified for blocking.	.sblock topic

In addition to the assembly directives that you can use in your code, the compiler produces several directives when it creates assembly code. These directives are to be used only by the compiler; do not attempt to use these directives.

- DWARF directives listed in Section A.1
- COFF/STABS directives listed in Section A.2
- The **.template** directive is used for early template instantiation. It encodes information about a template that has yet to be instantiated. This is a COFF C++ directive.
- The **.compiler_opts** directive indicates that the assembly code was produced by the compiler, and which build model options were used for this file.

4.2 Directives That Define Sections

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

- The **.bss** directive reserves space in the .bss section for uninitialized variables.
- The **.clink** directive enables conditional linking by telling the linker to leave the named section out of the final object module output of the linker if there are no references found to any symbol in the section. The .clink directive can be applied to initialized sections.
- The **.data** directive identifies portions of code in the .data section. The .data section usually contains initialized data.
- The **.sect** directive defines an initialized named section and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- The **.text** directive identifies portions of code in the .text section. The .text section usually contains executable code.
- The **.usect** directive reserves space in an uninitialized named section. The .usect directive is similar to the .bss directive, but it allows you to reserve space separately from the .bss section.

Chapter 2 discusses these 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	contains basic adding and loading instructions.
.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 000000                .text
 5 000000 3CA0           MOV #10,AC0
 6 000002 2201           MOV AC0,AC1
 7
 8              ************************************************
 9              *      Start assembling into the .data section   *
10              ************************************************
11 000000                .data
12 000000 0009           .word 9, 10
   000001 000A
13 000002 000B           .word 11, 12
   000003 000C
14
15              ************************************************
16              *      Start assembling into a named,            *
17              *      initialized section, var_defs             *
18              ************************************************
19 000000                .sect "var_defs"
20 000000 0011           .word 17, 18
   000001 0012
21
22              ************************************************
23              *     Resume assembling into the .data section   *
24              ************************************************
25 000004                .data
26 000004 000D           .word 13, 14
   000005 000E
27 000000                .bss sym, 19   ; Reserve space in .bss
28 000006 000F           .word 15, 16   ; Still in .data
   000007 0010
29
30              ************************************************
31              *    Resume assembling into the .text section    *
32              ************************************************
33 000004                .text
34 000004 2412           ADD AC1,AC2
35 000000      usym      .usect "xy", 20  ; Reserve space in xy
36 000006 2220           MOV AC2,AC0      ; Still in .text
```

4.3 Directives That Initialize Constants

Use These Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of these directives in a section that includes C55x instructions will likely lead to the generation of an invalid access to the data at execution. Consequently, Texas Instruments highly recommends that these directives be issued only within data sections.

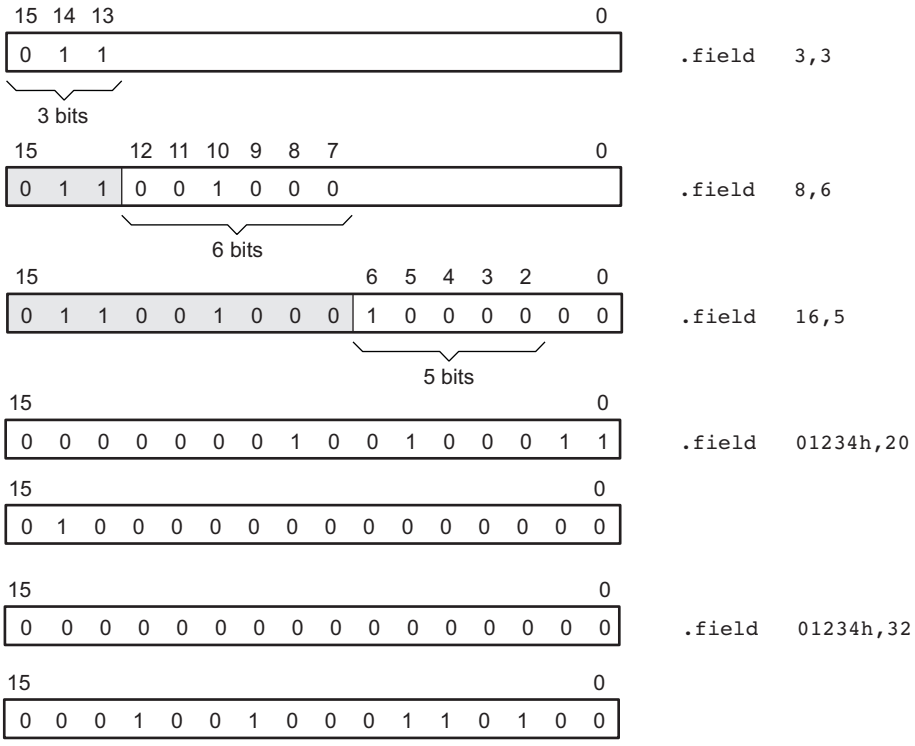
Several directives assemble values for the current section:

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

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

```
3 000000                .data
4 000000 6000           .field   3, 3
5 000000 6400           .field   8, 6
6 000000 6440           .field   16, 5
7 000001 0123           .field   01234h,20
  000002 4000
8 000003 0000           .field   01234h,32
  000004 1234
```

Figure 4-1. The .field Directive



- The **.float** and **.xfloat** directives calculate the single-precision (32-bit) IEEE floating-point representation of a single floating-point value and store it in a word in the current section that is aligned to a word boundary.
- The **.half**, **.uhalf**, **.int**, **.uint**, **.short**, **.ushort**, **.word**, and **.uword** directives place one or more 16-bit values into consecutive 16-bit fields (words) in the current section. The .int and .word directives automatically align to a word boundary.
- The **.ivec** directive is used to initialize the entries in the interrupt vector table.
- The **.long**, **.ulong**, and **.xlong** directives place one or more 32-bit values into consecutive 32-bit fields (words) in the current section. The most significant word is stored first. The .long directive automatically aligns to a long-word boundary; the .xlong directive does not.
- The **.string** , **.cstring**, and **.pstring** directives place 8-bit characters from one or more character strings into the current section. The .string and .cstring directives are similar to .byte, placing an 8-bit character in each consecutive word of the current section. The .cstring directive adds a NUL character needed by C; the .string directive does not add a NUL character. The .pstring directive also has a width of 8 bits, but it packs one character per byte. For .pstring, the last word in a string is padded with null characters (0) if necessary.

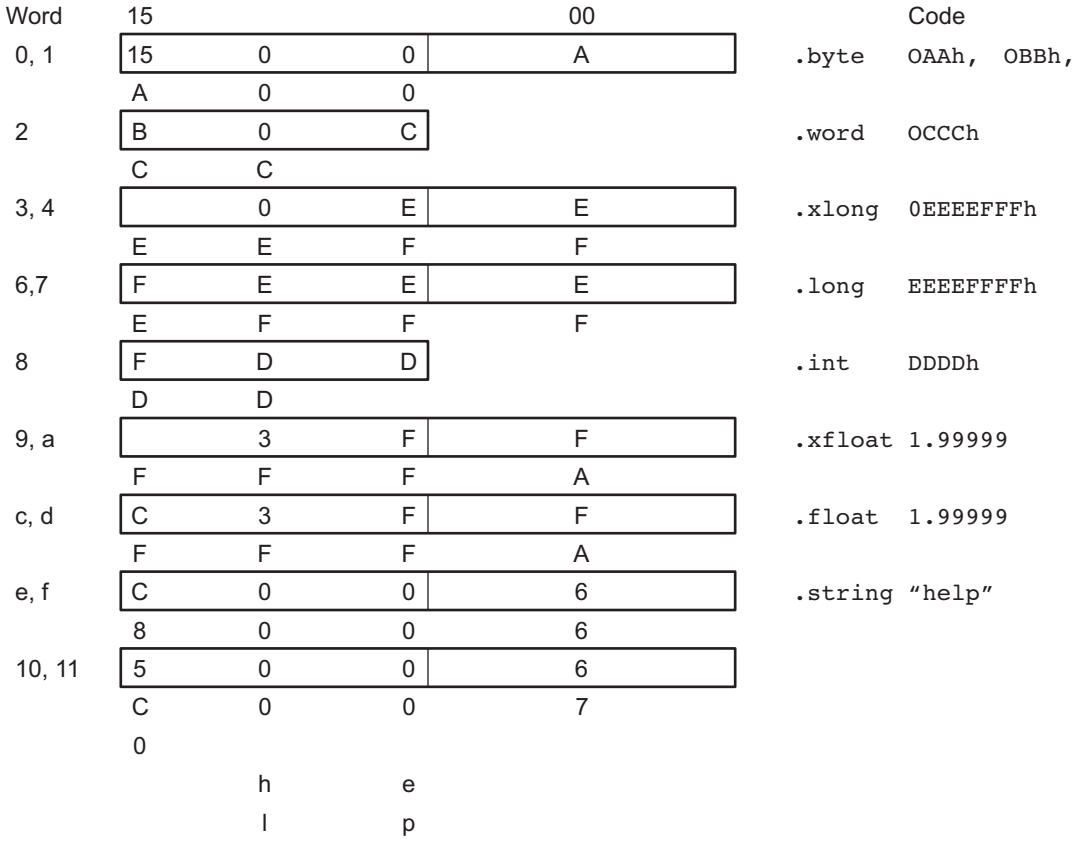
Directives That Initialize Constants When Used in a .struct/.endstruct Sequence

NOTE: The .byte, .char, .double, field, .float, .half, .int, .long, .short, .string, .ubyte, .uchar, .uhalt, .uint, .ulong, .ushort, .uword, and .word directives do not initialize memory when they are part of a .struct/ .endstruct sequence; rather, they define a member's size. For more information, see the .struct/.endstruct directives.

Figure 4-2 compares the .byte, .int, .long, .xlong, .float, .xfloat, .word, and .string directives. Using the following assembled code:

```
1 000000                .data
2 000000 00AA           .byte      0AAh, 0BBh
  000001 00BB
3 000002 0CCC           .word      0CCCh
4 000003 0EEE           .xlong     0EEEEFFFh
  000004 EFFF
5 000006 EEEE           .long      0EEEEFFFFh
  000007 FFFF
6 000008 DDDD           .int       0DDDDh
7 000009 3FFF           .xfloat    1.99999
  00000a FFAC
8 00000c 3FFF           .float     1.99999
  00000d FFAC
9 00000e 0068           .string    "help"
  00000f 0065
  000010 006c
  000011 0070
```

Figure 4-2. Initialization Directives



4.4 Directives That Perform Alignment and Reserve Space

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

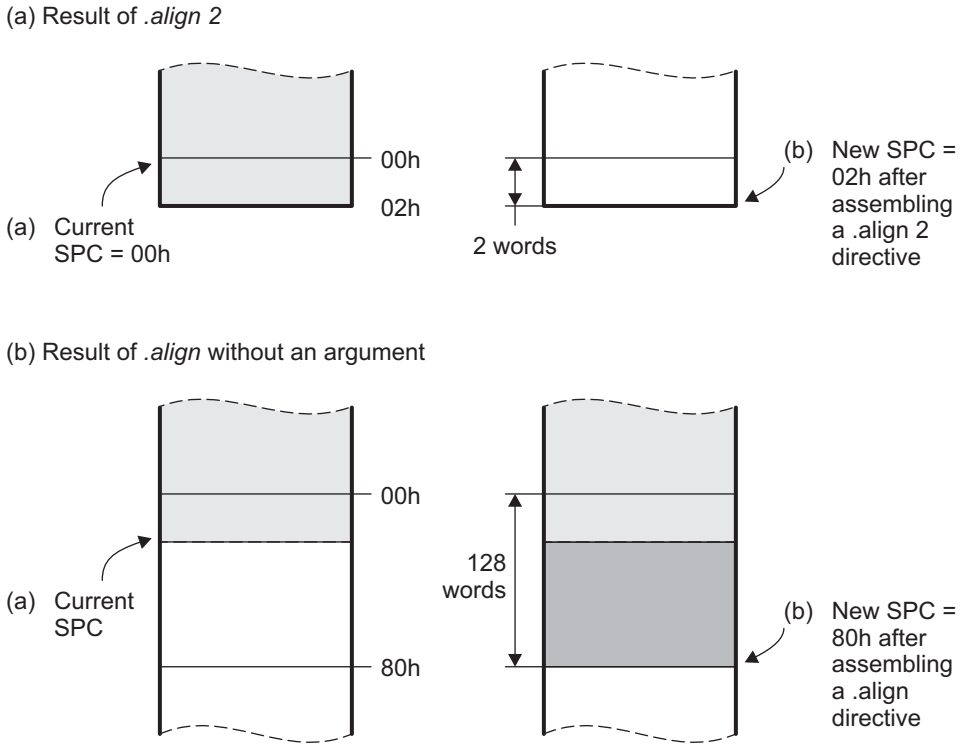
- The **.align** directive aligns the SPC at a byte boundary in code sections or a word boundary in data sections. If the SPC is already aligned at the selected boundary, it is not incremented. Operands for the .align directive must equal a power of 2 between 2^0 and 2^{16}.

  The .align directive with no operands defaults to a 128-byte boundary in a code section, and a 128-word (page) boundary in a data section.

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

```
 1 000000              .data
2 000000 4000         .field   2, 3
3 000000 4160         .field   11, 8
4                     .align   2
5 000002 0045         .string  "Errorcnt"
  000003 0072
  000004 0072
  000005 006f
  000006 0072
  000007 0063
  000008 006e
  000009 0074
6                     .align
7 000080 0004         .word    4
```

Figure 4-3. The .align Directive



- The **.even** directive aligns the SPC so that it points to the next word (in code sections) or long word (in data sections) boundary. It is equivalent to specifying the .align directive with an operand of 2. Any unused bits in the current byte or word are filled with 0s.
- The **.localalign** directive allows the maximum localrepeat loop size for the specified loop.
- The **.space** directive reserves a specified number of bits in the current section. The assembler fills these reserved bits with 0s.

  You can reserve words by multiplying the desired number of words by 16.

  When you use a label with .space, it points to the *first* byte that contains reserved bits.

  Assume the following code has been assembled:

```
1
2            ** .space directive
3 000000               .data
4 000000 0100          .word    100h, 200h
  000001 0200
5 000002     Res_1:    .space   17
6 000004 000F          .word    15
7            ** reserve 3 words
8 000005     Res_3:    .space   3*16
9 000008 000A          .word    10
```

  Res_1 points to the first word in the space reserved by .space.

4.5 Directives That Format the Output Listings

These directives format the listing file:

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

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

- The source code listing includes false conditional blocks that do not generate code. The **.fclist** and **.fcnolist** directives turn this listing on and off. You can use the .fclist directive to list false conditional blocks exactly as they appear in the source code. You can use the .fcnolist directive to list only the conditional blocks that are actually assembled.
- The **.length** directive controls the page length of the listing file. You can use this directive to adjust listings for various output devices.
- The **.list** and **.nolist** directives turn the output listing on and off. You can use the .nolist directive to prevent the assembler from printing selected source statements in the listing file. Use the .list directive to turn the listing on again.
- The source code listing includes macro expansions and loop blocks. The **.mlist** and **.mnolist** directives turn this listing on and off. You can use the .mlist directive to print all macro expansions and loop blocks to the listing, and the .mnolist directive to suppress this listing.
- The **.option** directive controls certain features in the listing file. This directive has the following operands:

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

- The **.page** directive causes a page eject in the output listing.
- The source code listing includes substitution symbol expansions. The **.sslist** and **.ssnolist** directives turn this listing on and off. You can use the .sslist directive to print all substitution symbol expansions to the listing, and the .ssnolist directive to suppress this listing. These directives are useful for debugging the expansion of substitution symbols.
- The **.tab** directive defines tab size.
- The **.title** directive supplies a title that the assembler prints at the top of each page.
- The **.width** directive controls the page width of the listing file. You can use this directive to adjust listings for various output devices.

4.6 Directives That Reference Other Files

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

- The **.copy** and **.include** directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from a copied file are printed in the listing file; the statements read from an included file are *not* printed in the listing file.
- The **.def** directive identifies a symbol that is defined in the current module and that can be used in another module. The assembler includes the symbol in the symbol table.
- The **.global** directive declares a symbol external so that it is available to other modules at link time. (For more information about global symbols, see Section 2.8.1). The .global directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. The linker resolves an undefined global symbol reference only if the symbol is used in the program. The .global directive declares a 16-bit symbol.
- The **.mlib** directive supplies the assembler with the name of an archive library that contains macro definitions. When the assembler encounters a macro that is not defined in the current module, it searches for it in the macro library specified with .mlib.
- The **.ref** directive identifies a symbol that is used in the current module but is defined in another module. The assembler marks the symbol as an undefined external symbol and enters it in the object symbol table so the linker can resolve its definition. The .ref directive forces the linker to resolve a symbol reference.
- The **.symdepend** directive creates an artificial reference from the section defining the source symbol name to the destination symbol. The .symdepend directive prevents the linker from removing the section containing the destination symbol if the source symbol section is included in the output module.

4.7 Directives That Enable Conditional Assembly

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

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

.if *well-defined expression*	marks the beginning of a conditional block and assembles code if the .if *well-defined expression* is true.
[**.elseif** *well-defined expression*]	marks a block of code to be assembled if the .if *well-defined expression* is false and the .elseif condition is true.
.else	marks a block of code to be assembled if the .if *well-defined expression* is false and any .elseif conditions are 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 [*well-defined expression*]	marks the beginning of a repeatable block of code. The optional expression evaluates to the loop count.
.break [*well-defined expression*]	tells the assembler to assemble repeatedly when the .break *well-defined expression* is false and to go to the code immediately after .endloop when the expression is true or omitted.
.endloop	marks the end of a repeatable block.

The assembler supports several relational operators that are useful for conditional expressions. For more information about relational operators, see Section 3.10.4.

4.8 Directives That Define Union or Structure Types

These directives set up specialized types for later use with the .tag directive, allowing you to use symbolic names to refer to portions of a complex object. The types created are analogous to the struct and union types of the C language.

The .struct, .union, .cstruct, and .cunion directives group related data into an aggregate structure which is more easily accessed. These directives do not allocate space for any object. Objects must be separately allocated, and the .tag directive must be used to assign the type to the object.

```
       .data
type   .struct           ; structure tag definition
X      .int
Y      .int
T_LEN  .endstruct
COORD  .tag type         ; declare COORD (coordinate)
       .bss COORD, T_LEN ; actual memory allocation
       .text
       ADD @(COORD.Y),AC0,AC0
```

The .cstruct and .cunion directives guarantee that the data structure will have the same alignment and padding as if the structure were defined in analogous C code. This allows structures to be shared between C and assembly code. See Chapter 14. For .struct and .union, element offset calculation is left up to the assembler, so the layout may be different than .cstruct and .cunion.

4.9 Directives That Define Enumerated Types

These directives set up specialized types for later use in expressions allowing you to use symbolic names to refer to compile-time constants. The types created are analogous to the enum type of the C language. This allows enumerated types to be shared between C and assembly code. See Chapter 14.

See Section 14.2.10 for an example of using .enum.

4.10 Directives That Define Symbols at Assembly Time

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

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

```
.asg "10, 20, 30, 40", coefficients
        ; Assign string to substitution symbol.
.byte coefficients
        ; Place the symbol values 10, 20, 30, and 40
        ; into consecutive bytes in current section.
```

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

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

- The **.define** 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 created with .define cannot be redefined.
- The **.label** directive defines a special symbol that refers to the load-time address within the current section. This is useful when a section loads at one address but runs at a different address. For example, you may want to load a block of performance-critical code into slower off-chip memory to save space and move the code to high-speed on-chip memory to run. See the .label topic for an example using a load-time address label.

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

```
bval .set 0100h   ; Set bval = 0100h
     .long bval, bval*2, bval+12
        ; Store the values 0100h, 0200h, and 010Ch
        ; into consecutive words in current section.
```

  The .set and .equ directives produce no object code. The two directives are identical and can be used interchangeably.
- The **.unasg** directive turns off substitution symbol assignment made with .asg.
- The **.undefine** directive turns off substitution symbol assignment made with .define.
- The **.var** directive allows you to use substitution symbols as local variables within a macro.

4.11 Directives That Communicate Run-Time Environment Details

These directives affect assembler assumptions while processing code. Within the ranges marked by these directives the assembler's default actions are altered as specified.

- The **.dp** directive specifies the value of the DP register. The assembler cannot track the value of the DP register; however, it needs to know the value of DP in order to assemble direct memory access operands. Consequently, this directive should be placed immediately following any instruction that changes the DP register's value. If the assembler is not given any information on the value of the DP register, it assumes the value is 0 when encoding direct memory operands.
- The **.lock_on** directive begins a block of code in which the assembler allows the lock() modifier. The **.lock_off** directives ends this block of code and resumes the default behavior of the assembler.
- The **.vli_off** directive begins a block of code in which the assembler uses the largest (P24) forms of certain variable-length instructions. By default, the assembler tries to resolve variable-length instructions to their smallest form. The **.vli_on** directive ends this block of code and resumes the default behavior of the assembler.

The following directives relate to C55x addressing modes:

- The **.arms_on** directive begins a block of code for which the assembler will use indirect access modifiers targeted to code size optimization. These modifiers are short offset modifiers. The **.arms_off** directive ends the block of code.
- The **.c54cm_on** directive signifies to the assembler that the following block of code has been converted from C54x code. The **.c54cm_off** directive ends the block of code.
- The **.cpl_on** directive begins a block of code in which direct memory addressing (DMA) is relative to the stack pointer. By default, DMA is relative to the data page. The **.cpl_off** directive ends the block of code.

The following directives relate to porting C54x code:

- The **.port_for_speed** directive begins a block of code in which the assembler encodes ported C54x code with a goal of achieving fast code. By default, the assembler encodes C54x code with a goal of achieving small code size. The **.port_for_size** directive ends the block of code.
- The **.sst_off** directive begins a block of code for which the assembler will assume that the SST status bit is set to 0. By default, the assembler assumes that the SST bit is set to 1. The **.sst_on** directive ends the block of code.

4.12 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler --symdebug:dwarf (-g) option to generate debug information for assembly functions.
- The **.cdecls** directive enables programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between C and assembly code.
- The **.end** directive terminates assembly. If you use the .end directive, it should be the last source statement of a program. This directive has the same effect as an end-of-file character.
- The **.newblock** directive resets local labels. Local labels are symbols of the form $n, where n is a decimal digit, or of the form NAME?, where you specify NAME. They are defined when they appear in the label field. Local labels are temporary labels that can be used as operands for jump instructions. The .newblock directive limits the scope of local labels by resetting them after they are used. See Section 3.9.2 for information on local labels.
- The **.noremark** directive begins a block of code in which the assembler suppresses the specified assembler remark. A remark is an informational assembler message that is less severe than a warning. The **.remark** directive re-enables the remark(s) previously suppressed by .noremark.
- The **.sblock** directive designates sections for blocking. A blocked section does not cross a page boundary (64 words) if it is smaller than a page, and it starts on a page boundary if it is larger than a page. Only initialized sections can be specified for blocking.
- The **.warn_on/.warn_off** directives enable and disable the issuing of warning messages by the assembler. By default, warnings are enabled (.warn_on).

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

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

For more information about using the error and warning directives in macros, see Section 5.7.

4.13 Directives Reference

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

.align *Align SPC on the Next Boundary*

Syntax

.align [*size in bytes*]

Description

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size in bytes* parameter. The *size* can be any power of 2, although only certain values are useful for alignment.

The size parameter should be in bytes for a code section, and in words for a data section. If a size is not specified, the SPC is aligned on the next 128-byte boundary for a code section, or the next 128-word (page) boundary for a data section.

A hole may be created by the .align directive if the SPC, at the point at which the directive occurs, is not on the desired byte or word boundary. In a data section, the assembler zero-fills holes created by .align. In a code section, holes are filled with NOP instructions.

The .even directive aligns the SPC on a word (code section) or long word (data section) boundary. This directive is equivalent to the .align directive with an operand of 2.

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 .even, .align 4, and a default .align.

```
 1 000000                .data
 2 000000 0004           .word     4
 3                       .even
 4 000002 0045           .string   "Errorcnt"
   000003 0072
   000004 0072
   000005 006F
   000006 0072
   000007 0063
   000008 006E
   000009 0074
 5                       .align
 6 000080 6000           .field    3,3
 7 000080 6A00           .field    5,4
 8                       .align    2
 9 000082 6000           .field    3,3
10                       .align    8
11 000088 5000           .field    5,4
12                       .align
13 000100 0004           .word     4
```

.arms_on/.arms_off *Display Code at Selected Address*

Syntax

.arms_on

.arms_off

Description

The **.arms_on** and **.arms_off** directives model the ARMS status bit.

The assembler cannot track the value of the ARMS status bit. You must use the assembler directives and/or command line options to communicate the value of this mode bit to the assembler. An instruction that modifies the value of the ARMS status bit should be immediately followed by the appropriate assembler directive.

The .arms_on directive models the ARMS status bit set to 1; it is equivalent to using the -ma command line option. The .arms_off directive models the ARMS status bit set to 0. In the case of a conflict between the command line option and the directive, the directive takes precedence.

By default (.arms_off), the assembler uses indirect memory access modifiers targeted to the assembly code.

In ARMS mode (.arms_on), the assembler uses short offset modifiers for indirect memory access. These modifiers are more efficient for code size optimization.

The scope of the .arms_on and .arms_off directives is static and not subject to the control flow of the assembly program. All assembly code between the .arms_on line and the .arms_off line is assembled in ARMS mode.

.asg/.define/.eval *Assign a Substitution Symbol*

Syntax

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

.define "*character string*"**,***substitution symbol*

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

Description

The **.asg** and **.define** directives assign 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 *substitution symbol* must be a valid symbol name. The substitution symbol is up to 32 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

The **.define** directive functions in the same manner as the .asg directive, except that .define disallows creation of a substitution symbol that has the same name as a register symbol or mnemonic. It does not create a new symbol name space in the assembler, rather it uses the existing substitution symbol name space. The .define directive is used to prevent corruption of the assembly environment when converting C/C++ headers. See Chapter 14 for more information about using C/C++ headers in assembly source.

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

- The *well-defined expression* is an alphanumeric expression in which all symbols have been previously defined in the current source module, so that the result is an absolute.
- The *substitution symbol* must be a valid symbol name. The substitution symbol is up to 32 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

See the .unasg/.undefine topic for information on turning off a substitution symbol.

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 000000 7b00        ADD #100,AC0
        000002 6400
      9 000004 b403        AMAR (*FP+)
#                          AMAR (AR0+)
     10
     11
     12 000000             .data
     13                    .asg 0, x
     14                    .loop 5
     15                    .eval x+1, x
     16                    .word x
     17                    .endloop
1                          .eval x+1, x
#                          .eval 0+1, x
1       000000 0001        .word x
#                          .word 1
1                          .eval x+1, x
#                          .eval 1+1, x
1       000001 0002        .word x
#                          .word 2
1                          .eval x+1, x
#                          .eval 2+1, x
1       000002 0003        .word x
#                          .word 3
1                          .eval x+1, x
#                          .eval 3+1, x
1       000003 0004        .word x
#                          .word 4
1                          .eval x+1, x
#                          .eval 4+1, x
1       000004 0005        .word x
#                          .word 5
```

.asmfunc/.endasmfunc *Mark Function Boundaries*

Syntax

symbol **.asmfunc** [**stack_usage(***num***)**]

.endasmfunc

Description

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

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

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

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

The .asmfunc directive has an optional parameter, stack_usage, which sets the stack to *num* bytes.

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

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

Example

In this example the assembly source generates debug information for the user_func section.

```
 1 000000                   .sect   ".text"
 2                          .align  4
 3                          .global userfunc
 4                          .global _printf
 5
 6              user_func:  .asmfunc
 7 000000 4EFD              AADD #-3, SP
 8 000002 FB00              MOV #(SL1 & 0xffff), *SP(#0)
   000004 0000%
 9 000006 6C00              CALL #_printf
   000008 0000!
10 00000a 3C04              MOV #0, T0
11 00000c 4E03              AADD #3, SP
12 00000e 4804              RET
13                          .endasmfunc
14
15 000000                   .sect   ".const"
16 000000 0048 SL1:         .string "Hello World!",10,0
   000001 0065
   000002 006C
   000003 006C
   000004 006F
   000005 0020
   000006 0057
   000007 006F
   000008 0072
   000009 006C
   00000a 0064
   00000b 0021
   00000c 000A
   00000d 0000
```

.bss *Reserve Space in the .bss Section*

Syntax

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

Description

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

- The *symbol* is a required parameter. It defines a label that points to the first location reserved by the directive. The symbol name must correspond to the variable that you are reserving space for.
- The *size in bytes* is a required parameter; it must be an absolute expression. The assembler allocates size bytes in the .bss section. There is no default size.
- The *blocking flag* is an optional parameter. If you specify a non-zero value for the parameter, the assembler reserves size words contiguously. This means that the reserved 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* is an optional parameter. The alignment is a power of two that specifies that the space reserved by this .bss directive is to be aligned to the specified word address boundary.

Specifying an Alignment Flag Only

NOTE: To specify an alignment flag without a blocking flag, you either insert two commas before the alignment flag, or specify 0 for the blocking flag.

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

Rule 1 Whenever a hole is left in memory (as shown in Figure 4-4), 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.

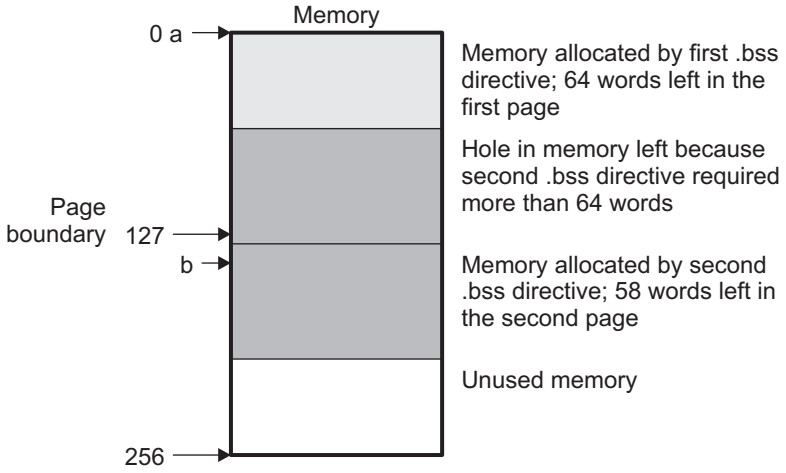
- If you do not request blocking, the memory is allocated at the current SPC.
- If you request blocking, the assembler checks to see whether there is enough space between the current SPC and the page boundary. If there is not enough space, the assembler creates another hole and allocates the space at the beginning of 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 4-4 shows, the second block is allocated on the next page.

Figure 4-4. 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 uses the .bss directive to reserve space in the .bss section, but then resumes assembling code into the current section (after the .bss has been processed). For more information about sections, see Chapter 2.

Example

In this example, the .bss directive reserves 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 placed contiguously within a page. Symbols declared with the .bss directive can be referenced in the same manner as other symbols and can also be declared external using the .global directive.

```
 1                ****************************************
 2                **   Assemble into the .text section.    **
 3                ****************************************
 4 000000                .text
 5 000000 3C00           MOV #0,AC0
 6                ****************************************
 7                **  Allocate 4 words in .bss for TEMP.   **
 8                ****************************************
 9 000000        Var_1:  .bss TEMP, 4
10                ****************************************
11                **            Still in .text            **
12                ****************************************
13 000002 7B00          ADD  #86,AC0,AC0
   000004 5600
14 000006 5272          MOV  T3,HI(AC2)
15 000008 1E73          MPYK #115,AC2,AC0
   00000a 80
16                ****************************************
17                **  Allocate 100 words in .bss for the   **
18                **  symbol named ARRAY; this part of     **
19                **  .bss must fit on a single page.      **
20                ****************************************
21 0000004              .bss ARRAY, 100, 1
22                ****************************************
23                **  Assemble more code into .text.       **
24                ****************************************
25 00000b C000-         MOV AC0,Var_1
26                ****************************************
27                **  Declare external .bss symbols.       **
28                ****************************************
29                      .global ARRAY, TEMP
30                      .end
```

.byte/.ubyte/.char/.uchar *Initialize Byte*

Syntax

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

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

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

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

Description

The **.byte**, **.ubyte**, **.char**, and **.uchar** directives place one or more 8-bit values into consecutive words in the current data section.

Use These Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of .byte, .ubyte, .char, and .uchar directives in a section that includes C55x instructions will lead to an invalid access to the data at execution. Consequently, it is highly recommended that these directives be used only in data sections.

In data sections, each 8-bit value is placed in a word by itself; the eight MSBs are filled with 0s. A value can be:

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

Values are not packed or sign-extended. In word-addressable data sections, each byte occupies the 8 least significant bits of a full 16-bit word. The assembler truncates values greater than 8 bits.

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

When you use these directives in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. For more information, see the .struct/.endstruct/.tag topic.

Example

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

```
1 000000             .data
2 000000             .space  100h * 16
3 000100 000a STRX   .byte   10, -1, "abc", 'a'
  000101 00ff
  000102 0061
  000103 0062
  000104 0063
  000105 0061
```

.c54cm_on/.c54cm_off *Display Code at Selected Address*

Syntax

.c54cm_on

.c54cm_off

Description

The **.c54cm_on** and **.c54cm_off** directives signify that a region of code has been converted from C54x code. The .c54cm_on and .c54cm_off directives model the C54CM status bit. The .c54cm_on directive models the C54CM status bit set to 1; it is equivalent to using the -ml command line option. The .c54cm_off directive models the C54CM status bit set to 0. In the case of a conflict between the command line option and the directive, the directive takes precedence.

The scope of the .c54cm_on and .c54cm_off directives is static and not subject to the control flow of the assembly program. All assembly code between the .c54cm_on and .c54cm_off directives is assembled in C54x compatibility mode.

In C54x compatibility mode, AR0 is used instead of T0 in memory operands. For example, *(AR5 + T0) is invalid in C54x compatibility mode; *(AR5 + AR0) should be used.

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

Syntax **Single Line:**

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

Syntax **Multiple Lines:**

.cdecls [*options*]

%{

/*--*/

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

/*--*/

%}

Description

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

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

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

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

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

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

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

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

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

See Chapter 14 for more information on setting up and using the .cdecls directive with C header files.

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

C header file:

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

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

struct myCstruct { int member_a; float member_b; };

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

Source file:

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

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

Listing File:

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

```
A     20          00000004 member_b         .field 32
      21                   ; float member_b - offset 4 bytes, size (4 bytes|32 bits)
A     22          00000008                  .endstruct
      23                   ; final size=(8 bytes|64 bits)
A     24
A     25                   ; ========== EXTERNAL FUNCTIONS ==========
A     26                                     .global _cvt_integer
A     27
A     28                   ; ========== EXTERNAL VARIABLES ==========
A     29                                     .global _a_variable
       2 00000000 00000008 size:    .int $sizeof(myCstruct)
       3 00000004 00000000 aoffset: .int myCstruct.member_a
       4 00000008 00000004 boffset: .int myCstruct.member_b
       5 0000000c 00000001 okvalue: .int status_enum.OK
       6 00000010 00000100 failval: .int status_enum.FAILED
       7                            .if $defined(WANT_ID)
       8 00000014 0000004A id       .cstring NAME
         00000015 0000006F
         00000016 00000068
         00000017 0000006E
         00000018 0000000A
         00000019 00000000
       9                            .endif
```

.clink *Conditionally Leave Section Out of Object Module Output*

Syntax

.clink["*section name*"]

Description

The **.clink** directive enables conditional linking by telling the linker to leave a section out of the final object module output of the linker if there are no references found to any symbol in that section. The .clink directive can be applied to initialized sections.

The .clink directive applies to the current initialized section. It tells the linker to leave the section out of the final object module output of the linker if there are no references found in a linked section to any symbol defined in the specified section.

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

Example

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

```
 1 000000              .sect "Vars"
 2             ; Vars section is conditionally linked
 3                     .clink
 4
 5 000000 001A X:      .word 01Ah
 6 000001 001A Y:      .word 01Ah
 7 000002 001A Z:      .word 01Ah
 8 000000              .sect "Counts"
 9             ; Counts section is conditionally linked
10                     .clink
11
12 000000 001A Xcount: .word 01Ah
13 000001 001A Ycount: .word 01Ah
14 000002 001A Zcount: .word 01Ah
15             ; By default, .text is unconditionally linked
16 000000              .text
17             ; Reference to symbol X cause the Vars section
18             ; to be linked into the COFF output
19 000000 3C00         MOV #0,AC0
20 000002 C000+        MOV AC0,X
```

.copy/.include *Copy Source File*

Syntax

.copy "*filename*"

.include "*filename*"

Description

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

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

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

The *filename* is a required parameter that names a source file. It is enclosed in double quotes and must follow operating system conventions.

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

1. The directory that contains the current source file
2. Any directories named with the --include_path assembler option
3. Any directories specified by the C55X_A_DIR environment variable
4. Any directories specified by the C55X_C_DIR environment variable

For more information about the --include_path option and C55X_A_DIR, see Section 3.5. For more information about C55X_C_DIR, see the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.

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

Example 1

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

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

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

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

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

Listing file:

```
      1 000000          .data
      2 000000          .space 29
      3 .copy           "byte.asm"
A     1            ** In byte.asm
A     2 000001          .data
A     3 000002 0020     .byte 32,1+ 'A'
        000003 0042
A     4                 .copy "word.asm"
B     1            * In word.asm
B     2 000004          .data
B     3 000004 ABCD     .word 0ABCDh, 56q
        000005 002E
A     5            ** Back in byte.asm
A     5 000006 006A     .byte 67h + 3q
      4
      5            ** Back in original file
      6 000007 646F     .pstring "done"
        000008 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)	byte2.asm (first copy file)	word2.asm (second copy file)

`.data`
`.space 29`
`.include "byte2.asm"`
`** Back in original file`
`.string "done"`
	`** In byte2.asm`
	`.data`
	`.byte 32,1+ 'A'`
	`.include "word2.asm"`
	`** Back in byte2.asm`
	`.byte 67h + 3q`
		`** In word2.asm`
		`.data`
		`.word 0ABCDh, 56q`

Listing file:

```
      1 000000          .data
      2 000000          .space 29
      3                 .include "byte2.asm"
      4
      5            ** Back in original file
      6 000007 0064     .string "done"
        000008 006F
        000009 006E
        00000a 0065
```

.cpl_on/.cpl_off *Select Direct Addressing Mode*

Syntax

.cpl_on

.cpl_off

Description

The **.cpl_on** and **.cpl_off** directives model the CPL status bit.

The assembler cannot track the value of the CPL status bit; you must use the assembler directives and/or command line option to model this mode for the assembler. An instruction that modifies the value of the CPL status bit should be immediately followed by the appropriate assembler directive.

The .cpl_on directive asserts that the CPL status bit is set to 1. When the .cpl_on directive is specified before any other instructions or directives that define object code, it is equivalent to using the -mc command line option. The .cpl_off directive asserts that the CPL status bit is set to 0. In the case of a conflict between the command line option and the directive, the directive takes precedence.

The .cpl_on and .cpl_off directives take no arguments.

In CPL mode (.cpl_on), direct memory addressing is relative to the stack pointer (SP). The dma syntax is *SP(dma), where dma can be a constant or a link-time-known symbolic expression. The assembler encodes the value of dma into the output bits.

By default (.cpl_off), direct memory addressing (dma) is relative to the data memory local page pointer register (DP). The dma syntax is @dma, where dma can be a constant or a relocatable symbolic expression. The assembler computes the difference between dma and the value in the DP register and encodes this difference into the output bits.

The assembler cannot track the value of the DP register; however, it must assume a value for the DP in order to assemble direct memory access operands. Consequently, you must use the .dp directive to model the DP value for the assembler. Issue this directive immediately following any instruction that changes the value in the DP register.

The scope of the .cpl_on and .cpl_off directives is static and not subject to the control flow of the assembly program. All assembly code between the .cpl_on line and the .cpl_off line is assembled in CPL mode.

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

Syntax

[*stag*] **.cstruct**|**.cunion** [*expr*]
[mem_0] *element* [$expr_0$]
[mem_1] *element* [$expr_1$]
. . .
. . .
. . .
[mem_n] **.tag** *stag* [$expr_n$]
[mem_N] *element* [$expr_N$]
[*size*] **.endstruct**|**.endunion**
label **.tag** *stag*

Description

The **.cstruct** and **.cunion** directives have been added to support ease of sharing of common data structures between assembly and C code. The .cstruct and .cunion directives can be used exactly like the existing .struct and .union directives except that they are guaranteed to perform data layout matching the layout used by the C compiler for C struct and union data types.

In particular, the .cstruct and .cunion directives force the same alignment and padding as used by the C compiler when such types are nested within compound data structures.

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

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

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

- The *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. A .stag is optional for .struct, but is required for .tag.
- The *element* is one of the following descriptors: .byte, .char, .double, field, .float, .half, .int, .long, .short, .string, .ubyte, .uchar, .uhalt, .uint, .ulong, .ushort, .uword, and .word.. All of these except .tag are typical directives that initialize memory. Following a .struct directive, these directives describe the structure element's size. They do not allocate memory. A .tag directive is a special case because stag must be used (as in the definition of stag).
- The *expr* is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.
- The $expr_{n/N}$ is an optional expression for the number of elements described. This value defaults to 1. A .string element is considered to be one byte in size, and a .field element is one bit.
- The $mem_{n/N}$ is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.
- The *size* is an optional label for the total size of the structure.

.data *Assemble Into the .data Section*

Syntax

.data

Description

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

For more information about sections, see Chapter 2.

Example

In this example, code is assembled into the .data (word-addressable) and .text (byte-addressable) sections.

```
 1                ****************************************
 2                **  Reserve space in .data.           **
 3                ****************************************
 4 000000                .data
 5 000000                .space 0CCh
 6
 7                ****************************************
 8                **  Assemble into .text.              **
 9                ****************************************
10 000000                .text
11                INDEX   .set 0
12 000000 3C00          MOV #INDEX,AC0
13
14                ****************************************
15                **  Assemble into .data.              **
16                ****************************************
17 00000c                .data
18 00000d ffff Table: .word   -1    ; Assemble 16-bit
19                                  ; constant into .data
20 00000e 00ff          .byte   0FFh  ; Assemble 8-bit
21                                  ; constant into .data
22                ****************************************
23                **  Assemble into .text.              **
24                ****************************************
25 000002                .text
26 000002 D600          ADD Table,AC0,AC0
   000004 00"
27
28                ****************************************
29                **  Resume assembling into the .data  **
30                **  section at address 0Fh.           **
31                ****************************************
32 00000f                .data
```

.double/.ldouble *Initialize Double-Precision Floating-Point Value*

Syntax

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

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

Description

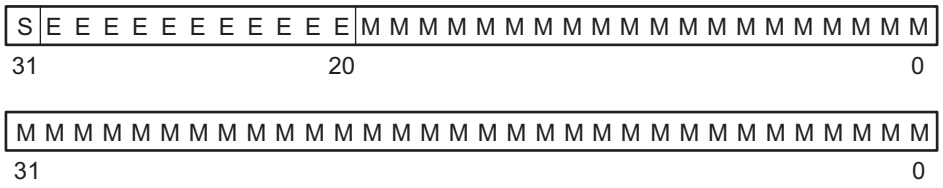
The **.double** and **.ldouble** directives place the IEEE double-precision floating-point representation of one or more floating-point values into the current section. Each *value* must be a floating-point constant or a symbol that has been equated to a floating-point constant. Each constant is converted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

Use .double and .ldouble Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of .double and .ldouble directives in a section that includes C55x instructions leads to an invalid access to the data at execution. Consequently, it is highly recommended that these directives be used only in data sections.

The 64-bit value is stored in the format shown in Figure 4-5.

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



Legend: S = sign
E = exponent (11-bit biased)
M = mantissa (52-bit fraction)

When you use .double or .ldouble in a .struct/.endstruct sequence, the directives define a member's size; they do not initialize memory. For more information, see the .struct/.endstruct/.tag topic.

Example

This example shows the .double and .ldouble directives.

```
1 000000             .data
2 000000 C520        .double   -1.0e25
  000001 8B2A
  000002 2C28
  000003 0291
2 000004 407C        .ldouble  456.0
  000005 8000
  000006 0000
  000007 0000
```

.dp *Specify DP Value*

Syntax

.dp *dp_value*

Description

The **.dp** directive specifies the value of the DP register. The *dp_value* can be a constant or a relocatable symbolic expression.

By default, direct memory addressing (dma) is relative to the data memory local page pointer register (DP). The dma syntax is @dma, where dma can be a constant or a relocatable symbolic expression. The assembler computes the difference between dma and the value in the DP register and encodes this difference into the output bits.

The assembler cannot track the value of the DP register; however, it must assume a value for the DP in order to assemble direct memory access operands. Consequently, you must use the .dp directive to model the DP value. Issue this directive immediately following any instruction that changes the value in the DP register. If the assembler is not informed of the value of the DP register, it assumes that the value is 0.

.drlist/.drnolist *Control Listing of Directives*

Syntax

.drlist

.drnolist

Description

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

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

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

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

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

Example

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

Source file:

```
        .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
      9                 .loop   3
     10                 .eval   x+1, x
     11                 .endloop
```

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

Syntax

.emsg *string*

.mmsg *string*

.wmsg *string*

Description

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

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

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

The **.wmsg** directive sends a warning message to the standard output device in the same manner as the .emsg directive. It increments the warning count rather than the error count, however. 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,AC0,AC0
        .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,AC0,AC0
      7                         .endif
      8                         .endm
      9
     10 000000                  MSG_EX PARAM
1                               .if $symlen(parm1) = 0
1                               .emsg "ERROR -- MISSING PARAMETER"
1                               .else
1       000000 D600             ADD  PARAM,AC0,AC0
        000002 00!
1                               .endif
     11
     12 000003                  MSG_EX
1                               .if $symlen(parm1) = 0
1                               .emsg "ERROR -- MISSING PARAMETER"
"emsg.asm", ERROR! at line 12: [***** USER ERROR ***** -]
ERROR -- MISSING PARAMETER
1                               .else
1                               ADD  parm1,AC0,AC0
```

```
                    1                           .endif
                    1 Error, No Warnings
```

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

```
"emsg.asm", ERROR! at line 12: [***** USER ERROR ***** -] ERROR -- MISSING PARAMETER
                  .emsg "ERROR -- MISSING PARAMETER"

 1 Error, No Warnings

Errors in source - Assembler Aborted
```

.end *End Assembly*

Syntax

.end

Description

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

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

Ending a Macro

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

Example

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

Source file:

```
        .data
START:  .space  300
TEMP    .set    15
        .bss    LOC1, 48h
        .data
        ABS AC0,AC0
        ADD #TEMP,AC0,AC0
        MOV AC0,LOC1
        .end
        .byte   4
        .word   CCCh
```

Listing file:

```
      1 000000                .data
      2 000000       START:   .space  300
      3              TEMP     .set    15
      4 000000                .bss    LOC1, 48h
      5 000000                .data
      5 000000 3200           ABS AC0,AC0
      6 000002 40F0           ADD #TEMP,AC0,AC0
      7 000004 C000-          MOV AC0,LOC1
      8                      .end
```

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

Syntax

.fclist

.fcnolist

Description

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

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

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

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

Example

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

Source file:

```
AAA    .set  1
BBB    .set  0
       .fclist
       .if   AAA
       ADD #1024,AC0,AC0
       .else
       ADD #(1024*10),AC0,AC0
       .endif

       .fcnolist
       .if   AAA
       ADD #1024,AC0,AC0
       .else
       ADD #(1024*10),AC0,AC0
       .endif
```

Listing file:

```
     1              AAA    .set  1
     2              BBB    .set  0
     3                     .fclist
     4                     .if   AAA
     5 000000 7B04         ADD #1024,AC0,AC0
     6                     .else
     7                     ADD #(1024*10),AC0,AC0
     8                     .endif
     9
    10                     .fcnolist
    11
    13 000004 7B04         ADD #(1024*10),AC0,AC0
       000006 0000
```

.field *Initialize Field*

Syntax

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

Description

The **.field** directive initializes a multiple-bit field within a single word (16 bits) of memory. This directive has two operands:

- The *value* is a required parameter; it is an expression that is evaluated and placed in the field. The value must be absolute.
- The *size in bits* is an optional parameter; it specifies a number from 1 to 32, which is the number of bits in the field. If you do not specify a size, the assembler assumes the size is 16 bits. If you specify a value that cannot fit in *size in bits*, the assembler truncates the value and issues a warning message. For example, .field 3,1 causes the assembler to truncate the value 3 to 1; the assembler also prints the message:

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

Use .field Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of the .field directive in a section that includes C55x instructions will lead to an invalid access to the data at execution. Consequently, it is highly recommended that these directives be used only in data sections.

Successive .field directives pack values into the specified number of bits.

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

If you use a label, it points to the 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, see the .struct/.endstruct/.tag topic.

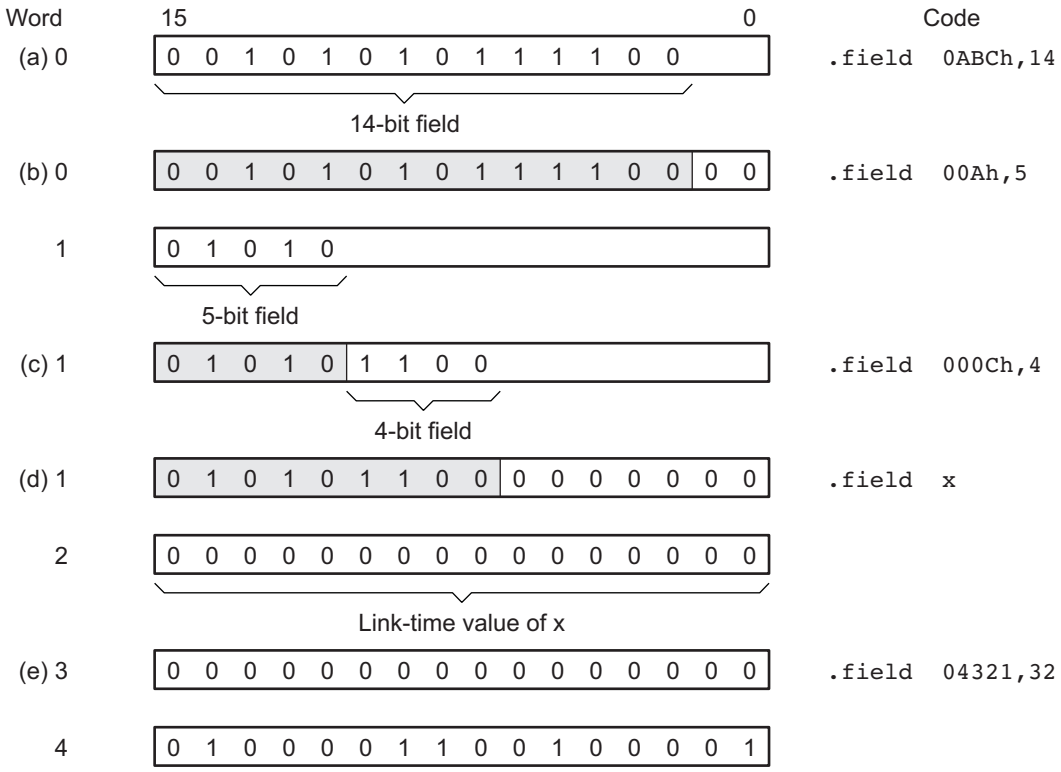
Example

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

```
 1 000000               .data
 2              ***********************************
 3              **   Initialize a 14-bit field.   **
 4              ***********************************
 5 000000 2AF0          .field 0ABCh, 14
 6
 7              ***********************************
 8              **    Initialize a 5-bit field    **
 9              **          in a new word.        **
10              ***********************************
11 000001 5000 L_F:    .field 0Ah, 5
12
13              ***********************************
14              **    Initialize a 4-bit field    **
15              **       in the same word.        **
16              ***********************************
17 000001 5600 x:      .field 0Ch, 4
18
19              ***********************************
20              **    16-bit relocatable field    **
21              **       in the next word.        **
22              ***********************************
23 000002 0001"        .field x
24
25              ***********************************
26              **   Initialize a 32-bit field.   **
```

```
27              ************************************
28 000003 0000          .field 04321h, 32
   000004 4321
```

Figure 4-6. The .field Directive



.float/.xfloat *Initialize Single-Precision Floating-Point Value*

Syntax

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

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

Description

The **.float** directive places the IEEE single-precision floating-point representation of a single floating-point constant into a word in 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.

Use .float and .xfloat Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of the .float and .xfloat directives in a section that includes C55x instructions will lead to an invalid access to the data at execution. Consequently, it is highly recommended that these directives be used only in data sections.

The 32-bit value is stored exponent byte first, most significant byte of fraction second, and least significant byte of fraction third, in the format shown in Figure 4-7.

Figure 4-7. Single-Precision Floating-Point Format

value = $(-1)^S$ x (1.0 + mantissa) x $(2)^{exponent-127}$

Legend: S = sign (1 bit)
E = exponent (8-bit biased)
M = mantissa (23-bit fraction)

When you use .float and .xfloat in a .struct/.endstruct sequence, .float and .xfloat define a member's size; they do not initialize memory. For more information, see the .struct/.endstruct/.tag topic.

Example

Following are examples of the .float directive:

```
1 000000              .data
2 000000 E904         .float -1.0e25
  000001 5951
3 000002 4040         .float 3
  000003 0000
4 000004 42F6         .float 123
  000005 0000
```

.global/.def/.ref *Identify Global Symbols*

Syntax

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

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

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

Description

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

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

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

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

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

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

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

Example

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

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

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

file1.lst

```
 1              ; Global symbol defined in this file
 2                        .global INIT
 3              ; Global symbols defined in file2.lst
 4                        .global X, Y, Z
 5 000000       INIT:
 6 000000 7B00            ADD #86,AC0,AC0
   000002 5600
 7 000000                 .data
 8 000000 0000!           .word   X
 9              ;    .
10              ;    .
11              ;    .
12                        .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              X:        .set    1
 6              Y:        .set    2
 7              Z:        .set    3
 8 000000                 .data
 9 000000 0000!           .word   INIT
10              ;    .
11              ;    .
12              ;    .
13                       .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 000000       INIT:
 6 000000 7B00            ADD #86,AC0,AC0
   000002 5600
 7 000000                 .data
 8 000000 0000!           .word   X
 9              ;    .
10              ;    .
11              ;    .
12                       .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              X:        .set 1
 6              Y:        .set 2
 7              Z:        .set 3
 8 000000                 .data
 9 000000 0000!           .word   INIT
10              ;    .
11              ;    .
12              ;    .
13                       .end
```

.half/.short/.uhalf/.ushort *Initialize 16-Bit Integers*

Syntax

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

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

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

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

Description

The **.half**, **.uhalf**, **.short**, and **.ushort** directives place one or more values into consecutive 16-bit fields in the current section. A *value* can be either:

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

Use These Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of the .half, .uhalf, .short, and .ushort directives in a section that includes C55x instructions will lead to an invalid access to the data at execution. Consequently, it is highly recommended that these directives be used only in data sections.

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.

The assembler truncates values greater than 16 bits.

If you use a label with .half, .short, .uhalf, or .ushort; it points to the location where the assembler places the first initialized word.

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

Example

In this example, the .half directive is used to place 16-bit values (10, -1, abc, and a) into memory; .short is used to place 16-bit values (8, -3, def, and b) into memory. The label STRN has the value 106h, which is the location of the first initialized word.

```
1 000000             .data
2 000000             .space 100h * 16
3
4 000100 000A        .half 10, -1, "abc", 'a'
  000101 FFFF
  000102 0061
  000103 0062
  000104 0063
  000105 0061
5 000106 0008 STRN   .short 8, -3, "def", 'b'
  000107 FFFD
  000108 0064
  000109 0065
  00010a 0066
  00010b 0062
```

.if/.elseif/.else/.endif *Assemble Conditional Blocks*

Syntax

.if *well-defined expression*

[**.elseif** *well-defined expression*]

[**.else**]

.endif

Description

Four directives provide conditional assembly:

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

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

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

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

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

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

See Section 3.10.4 for information about relational operators.

Example

This example shows conditional assembly:

```
 1              SYM1    .set    1
 2              SYM2    .set    2
 3              SYM3    .set    3
 4              SYM4    .set    4
 5 000000               .data
 6              If_4:   .if     SYM4 = SYM2 * SYM2
 7 000000 0004          .byte   SYM4 ; Equal values
 8                      .else
 9                      .byte   SYM2 * SYM2 ; Unequal values
10                      .endif
11              If_5:   .if SYM1 <= 10
12 000001 000a          .byte   10 ; Less than / equal
13                      .else
14                      .byte   SYM1 ; Greater than
15                      .endif
16              If_6:   .if     SYM3 * SYM2 != SYM4 + SYM2
17                      .byte   SYM3 * SYM2 ; Unequal value
18                      .else
19 000002 0008          .byte   SYM4 + SYM4 ; Equal values
20                      .endif
21              If_7:   .if SYM1 = 2
22                      .byte   SYM1
23                      .elseif SYM2 + SYM3 = 5
24 000003 0005          .byte SYM2 + SYM3
25                      .endif
```

.int/.word/.uint/.uword *Initialize 16-Bit Integers*

Syntax

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

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

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

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

Description

The **.int**, **.uint**, **.word**, and **.uword** directives place one or more values into consecutive words in the current section. Each value is placed in a 16-bit word by itself and is aligned on a word boundary. A *value* can be either:

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

Use These Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of the .int, .uint, .word, and .uword directives in a section that includes C55x instructions will lead to an invalid access to the data at execution. Consequently, it is highly recommended that these directives be used only in data sections.

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

If you use a label with these directives, it points to the first word that is initialized.

When you use these directives in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. See the .struct/.endstruct/.tag topic.

Example 1

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

```
1 000000             .data
2 000000             .space 73h
3 000000             .bss   PAGE, 128
4 000080             .bss   SYMPTR, 3
5 000000             .text
6 000000 7600 INST:  MOV #86,AC0
  000002 5608
7 000007             .data
8 000008 000A        .int   10, SYMPTR, -1, 35 + 'a'
  000009 0080-
  00000a FFFF
  00000b 0084
```

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 000000             .data
1 000000 0C80 WORDX: .word 3200, 1 + 'AB', -0AFh, 'X'
  000001 4143
  000002 FF51
  000003 0058
```

.ivec *Initialize Interrupt Table Entries*

Syntax

[*label*] **.ivec**[*address*[,*stack mode* | *configuration value*]]

Description

The **.ivec** directive is used to initialize an entry in the interrupt vector table.

This directive has the following operands:

- The *label*, if specified, will be assigned the code (byte) address associated with the directive, not the data (word) address as with other directives.
- The *address* specifies the address of the interrupt service routine. If an address is not specified, 0 is used.
- You can specify a *stack mode* only for the reset vector, which must be the first .ivec in the interrupt vector table. The stack mode can be identified as follows:

C54X_STK	This value specifies the 32-bit stack needed by converted C54x code. This is the default if no value is given for the stack mode.
NO_RETA	This value specifies 16-bit slow return mode.
USE_RETA	This value specifies 16-bit plus register fast return mode.

More information on the stack modes can be found in the *TMS320C55x DSP CPU Reference Guide*. You can write these symbolic names in either upper or lower case.

The .ivec directive aligns the SPC on an 8-byte boundary, so that you are not forced to place an instruction between two .ivec entries. Any space added for this alignment is filled with NOP instructions.

In general, a section that contains other data defining directives (such as .word) is characterized as a data section. A data section is word-addressable and cannot contain code. A section containing the .ivec directive is characterized as a code section (byte-addressable), and can include other instructions. Like an instruction, .ivec cannot be mixed with other data defining directives.

A section containing an .ivec directive is marked as an interrupt vector section. The linker can recognize such sections, and does not add a non-parallel NOP at the end of it, as it does for normal code sections.

Example

This example shows the use of the .ivec directive.

```
       .sect "vectors"          ; start vectors section
       .ref  start,nmi_isr,isr2 ; symbols referenced
                                ; from other files
       .def  rsv,no_isr         ; symbols defined in this
                                ; file
rsv:   .ivec start,c54x_stk     ; C54x compatibility
                                ; stack mode
nmi    .ivec nmi_isr            ; standard usage
int3   .ivec                    ; one way to skip a vector
int4   .ivec no_isr             ; better way to skip a vector
; ... and so on. Fill out all 32 vectors.
int31  .ivec no_isr             ; last vector
       .text                    ; change to text section
no_isr B     no_isr             ; default ISR
```

Note the difference between int3 and int4. If the int3 vector is raised, the example branches to 0, with unpredictable results. However, if the int4 vector is raised, the example branches to the no_isr spin loop, which generates predictable results.

.label *Create a Load-Time Address Label*

Syntax

.label *symbol*

Description

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

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

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

Example

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

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

See Section 9.6.5 for more information about assigning run-time and load-time addresses in the linker.

.length/.width *Set Listing Page Size*

Syntax

.length [*page length*]

.width [*page width*]

Description

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

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

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

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

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

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

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

Example

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

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

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

.list/.nolist *Start/Stop Source Listing*

Syntax

.list

.nolist

Description

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

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

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

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

By default, the source listing is printed to the listing file; the assembler acts as if the .list directive had been used. However, if you do not request a listing file when you invoke the assembler by including the --asm_listing option on the command line (see Section 3.3), 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. The .nolist, the second .copy, and the .list directives do not appear in the listing file. 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"
   .localalign
```

Listing file:

```
      1                    .copy "copy2.asm"
A     1              * In copy2.asm (copy file)
A     2 000000               .data
A     3 000000 0020          .word 32, 1 + 'A'
      4 000001 0042
      2              * Back in original file
      3 000000               .text
      4 000000 90            NOP
      9              * Back in original file
     10 000004               .data
     11 000004 0044          .string "Done"
        000005 006F
        000006 006E
        000007 0065
```

.localalign — *Create a Load-Time Address Label*

Syntax

.localalign

Description

The assembler directive **.localalign**, meant to be placed right before a localrepeat instruction, causes the first instruction in the body of the loop to be aligned to a 4-byte alignment, which allows the maximum localrepeat loop size. It operates by inserting enough single-cycle NOP instructions to get the alignment correct. It also causes a 4-byte alignment to be applied to the current section so the linker honors the necessary alignment for that loop body. It takes no parameters.

Example 1

This example shows the behavior of a localrepeat loop without the .localalign directive.

```
main:  nop
       nop
       nop
       localrepeat  {
          AC1 = #5
          AC2 = AC1 }
```

The above source code produces this output:

```
1 000000 20   main:  nop
2 000001 20          nop
3 000002 20          nop
4 000003 4A82        localrepeat  {
5 000005 3C51           AC1 = #5
6 000007 2212           AC2 = AC1 }
```

Example 2

This example shows the source code from Example 1 after .localalign is added.

```
main:  nop
       nop
       nop
       .localalign
       localrepeat  {
          ac1 = #5
          ac2 = ac1 }
```

This example produces an aligned loop body before the localrepeat on line 5, causing the loop body beginning at line 6 to now be 4-byte aligned; its address went from 0x5 to 0x8:

```
1 000000 20    main:  nop
2 000001 20           nop
3 000002 20           nop
4                     .localalign
5 000006 4A82         localrepeat  {
6 000008 3C51            AC1 = #5
7 00000a 2212            AC2 = AC1 }
```

A disassembly shows how NOPs were inserted:

```
TEXT Section .text, 0xC bytes at 0x0
000000: 20              NOP
000001: 20              NOP
000002: 20              NOP
000003: 5e80_21         NOP_16 || NOP
000006: 4a82            RPTBLOCAL 0xa
000008: 3c51            MOV #5,AC1
00000a: 2212            MOV AC1,AC2
```

By aligning the loop using the .localalign directive (or even by hand), the localrepeat loops can achieve maximum size. Without this alignment, the loops may need to be several bytes shorter due to how the instruction buffer queue (IBQ) on the C55x processor is loaded.

While the directive can be used with short loops, .localalign really only needs to be used on localrepeat loops that are near the limit of the localrepeat size.

.lock_on/.lock_off *Enable read-modify-write Instruction Range*

Syntax

.lock_on

.lock_off

Description

The **.lock_on** and **.lock_off** directives identify a range for use with read-modify-write instructions. Within this range, the lock() qualifier can be used with any read-modify-write instruction. If a lock() qualifier is not used with a read-modify-write instruction that exists in a .lock_on block, then the assembler will issue a remark diagnostic stating that the operation is not guaranteed to be atomic. Outside of the range of the .lock_on and .lock_off directives, the lock() qualifier is illegal and read-write-modify instructions are not flagged.

These directives are intended to be placed around a critical region of code where atomic access to a memory location must be guaranteed.

By default, the assembler treats all code as being outside of a .lock_on/.lock_off range.

.long/.ulong/.xlong *Initialize 32-Bit Integer*

Syntax

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

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

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

Description

The **.long**, **.ulong**, 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 and .ulong directives align the result on the long word boundary, while the .xlong directive does not. A value can be:

- An expression that the assembler evaluates and treats as a 32-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value.

Use These Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of the .long, .ulong, and .xlong directives in a section that includes C55x instructions will lead to an invalid access to the data at execution. Consequently, it is highly recommended that these directives be used only in data sections.

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

If you use a label with these directives, it points to the first word that is initialized.

When you use .long, .ulong, or .xlong directives in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. See the .struct/.endstruct/.tag topic.

Example

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

```
1 000000            .data
2 000000 0000 DAT1: .long  0ABCDh, 'A' + 100h, 'g', 'o'
  000001 ABCD
  000002 0000
  000003 0141
  000004 0000
  000005 0067
  000006 0000
  000007 006F
3 000008 0000       .xlong DAT1, 0AABBCCDDh
  000009 0000"
  00000a AABB
  00000b CCDD
4 00000c      DAT2:
```

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

Syntax

.loop [*well-defined expression*]

.break [*well-defined expression*]

.endloop

Description

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

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

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

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

Example

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

```
      1 000000            .data
      2                   .eval      0,x
      3              LAB_1 .loop
      4                   .word      x*100
      5                   .eval      x+1, x
      6                   .break     x = 6
      7                   .endloop
1       000000 0000       .word      0*100
1                         .eval      0+1, x
1                         .break     1 = 6
1       000001 0064       .word      1*100
1                         .eval      1+1, x
1                         .break     2 = 6
1       000002 00C8       .word      2*100
1                         .eval      2+1, x
1                         .break     3 = 6
1       000003 012C       .word      3*100
1                         .eval      3+1, x
1                         .break     4 = 6
1       000004 0190       .word      4*100
1                         .eval      4+1, x
1                         .break     5 = 6
1       000005 01F4       .word      5*100
1                         .eval      5+1, x
1                         .break     6 = 6
```

.macro/.endm *Define Macro*

Syntax

macname **.macro** [*parameter*$_1$[**, ... ,** *parameter*$_n$]]

model statements or macro directives

.endm

Description

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

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

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

Macros are explained in further detail in Chapter 5.

.mlib *Define Macro Library*

Syntax

.mlib "*filename*"

Description

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

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

1. The directory that contains the current source file
2. Any directories named with the --include_path assembler option
3. Any directories specified by the C55X_A_DIR environment variable
4. Any directories specified by the C55X_C_DIR environment variable

See Section 3.5 for more information about the --include_path option.

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

See Chapter 5 for more information on macros and macro libraries.

Example

The code creates a macro library that defines two macros, incr.asm and decr.asm. The file incr.asm contains the definition of incr and decr.asm contains the definition of decr.

incr.asm	decr.asm

`* Macro for incrementing`
`inc1 .macro A`
`ADD #1,AC0,AC0`
`ADD #1,AC1,AC1`
`ADD #1,AC2,AC2`
`ADD #1,AC3,AC3`
`.endm`
	`* Macro for decrementing`
	`dec1 .macro A`
	`SUB #1,AC0,AC0`
	`SUB #1,AC1,AC1`
	`SUB #1,AC2,AC2`
	`SUB #1,AC3,AC3`
	`.endm`

Use the archiver to create a macro library:

```
ar55 -a mac incr.asm decr.asm
```

Now you can use the .mlib directive to reference the macro library and define the incr.asm and decr.asm macros:

```
      1              .mlib   "mac.lib"
      2 000000       incr              ; Macro call
1       000000 4010     ADD #1,AC0,AC0
1       000002 4011     ADD #1,AC1,AC1
1       000004 4012     ADD #1,AC2,AC2
1       000006 4013     ADD #2,AC3,AC3
      3 000008       decr              ; Macro call
1       000008 4210     SUB #1,AC0,AC0
1       00000a 4211     SUB #1,AC1,AC1
1       00000c 4212     SUB #1,AC2,AC2
1       00000e 4213     SUB #1,AC3,AC3
```

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

Syntax

.mlist

.mnolist

Description

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

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

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

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

See Chapter 5 for more information on macros and macro libraries. See the .loop/.break/.endloop topic for information on conditional blocks.

Example

This example defines a macro named STR_3. The first time the macro is called, the macro expansion is listed (by default). The second time the macro is called, the macro expansion is not listed, because a .mnolist directive was assembled. The third time the macro is called, the macro expansion is again listed because a .mlist directive was assembled.

```
      1              STR_3  .macro P1, P2, P3
      2                     .data
      3                     .string ":p1:", ":p2:", ":p3:"
      4                     .endm
      5
      6 000000              STR_3 "as", "I", "am"
1       000000              .data
1       000000 003A         .string ":p1:", ":p2:", ":p3:"
        000001 0070
        000002 0031
        000003 003A
        000004 003A
        000005 0070
        000006 0032
        000007 003A
        000008 003A
        000009 0070
        00000a 0033
        00000b 003A
      7                     .mnolist
      8 00000c              STR_3 "as", "I", "am"
      9                     .mlist
     10 000018              STR_3 "as", "I", "am"
1       000018              .data
1       000018 003A         .string ":p1:", ":p2:", ":p3:"
        000019 0070
        00001a 0031
        00001b 003A
        00001c 003A
        00001d 0070
        00001e 0032
        00001f 003A
        000020 003A
        000021 0070
        000022 0033
        000023 003A
```

.newblock *Terminate Local Symbol Block*

Syntax

.newblock

Description

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

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

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

See Section 3.9.2 for more information on the use of local labels.

Example

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

```
 1                    .ref  ADDRA, ADDRB, ADDRC
 2             foo    .set  76h
 3
 4 000000 A000! LABEL1: MOV ADDRA,AC0
 5 000002 7C00          SUB #foo,AC0
   000004 7600
 6 000006 62200         BCC $1,AC0 < #0
 7 000008 A000!         MOV ADDRB,AC0
 8 00000a 4A02          B $2
 9
10 00000c A000! $1      MOV ADDRA,AC0
11 000003 D600  $2      ADD ADDRC,AC0,AC0
   000010 00!
12                      .newblock  ; Undefine $1 to reuse
13 000011 6120          BCC $1,AC0 < #0
14 000013 C000!         MOV AC0,ADDRC
15 000015 20    $1      NOP
```

.noremark/.remark *Control Remarks*

Syntax

.noremark *num*

.remark [*num*]

Description

The **.noremark** directive suppresses the assembler remark identified by num. A remark is an informational assembler message that is less severe than a warning. For a description of remarks, see Section 7.7.

This directive is equivalent to using the -ar[*num*] assembler option.

The **.remark** directive re-enables the remark(s) previously suppressed.

Example

This example shows how to suppress the R5002 remark:

Original listing file:

```
 1  000000 20              RSBX CMPT
"file.asm", REMARK  at line 1: [R5002] Ignoring RSBX CMPT instruction
 2
 3 000001 4804 RETF
"file.asm", REMARK  at line 3: [R5004] Translation of RETF correct
only for non-interrupt routine
```

Listing file with .noremark:

```
 1                      .noremark 5002
 2 000000 20            RSBX CMPT
 3
 4 000001 4804          RETF
"file.asm", REMARK at line 4: [R5004] Translation of RETF correct
only for non-interrupt routine
```

.option *Select Listing Options*

Syntax

.option *option*$_1$[**,** *option*$_2$**,**. . .]

Description

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

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

Options *are not* case sensitive.

Example

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

```
 1                ****************************************
 2                ** Limit the listing of .byte, .long, **
 3                **   .word, and .string directives    **
 4                **           to 1 line each.          **
 5                ****************************************
 6                            .option B, W, L, T
 7 000000                     .data
 8 000000 00BD                .byte   -'C', 0B0h, 5
 9 000004 AABB                .long   0AABBCCDDh, 536 + 'A'
10 000008 15AA                .word   5546, 78h
11 00000a 0045                .string "Extended Registers"
12
13                ****************************************
14                **     Reset the listing options.     **
15                ****************************************
16 .option R
17 00001c FFBD                .byte   -'C', 0B0h, 5
   00001d 00B0
   00001e 0005
18 000020 AABB                .long   0AABBCCDDh, 536 + 'A'
   000021 CCDD
   000022 0000
   000023 0259
19 000024 15AA                .word   5546, 78h
   000025 0078
20 000026 0045                .string "Extended Registers"
   000027 0078
   000028 0074
   000029 0065
   00002a 006E
   00002b 0064
   00002c 0065
```

```
00002d 0064
00002e 0020
00002f 0052
000030 0065
000031 0067
000032 0069
000033 0073
000034 0074
000035 0065
000036 0072
000037 0073
```

.page *Eject Page in Listing*

Syntax **.page**

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

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

Source file:

```
Source file (generic)
        .title   "**** Page Directive Example ****"
;         .
;         .
;         .
        .page
```

Listing file:

```
TMS320C55x Assembler    Version x.xx        Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE    1

       2                    ;        .
       3                    ;        .
       4                    ;        .
TMS320C55x Assembler    Version x.xx        Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE    2


 No Errors, No Warnings
```

.port_for_speed/.port_for_size *Encode C54x Instructions for Speed or Size*

Syntax

.port_for_speed

.port_for_size

Description

The **.port_for_speed** and **.port_for_size** directives affect the way the assembler encodes certain C54x instructions when ported to C55x. By default, masm55 tries to encode C54x instructions to achieve small code size (.port_for_size). Use .port_for_speed, or the −mh assembler option, to allow the assembler to generate a faster encoding. For more information, see Section 7.2.2.

The .port_for_size directive models the default encoding of the assembler.

The .port_for_speed directive models the effect of the -mh assembler option. In the case of a conflict between the command line option and the directive, the directive takes precedence.

Consider using .port_for_speed just before a critical loop. After the loop, use .port_for_size to return to the default encoding.

.sblock *Specify Blocking for an Initialized Section*

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. In a code section, blocked code is guaranteed not to cross a 128-byte boundary if it is smaller than 128 bytes, or to start on a 128-byte boundary if it is larger than 128 bytes. In a data section, blocked code is guaranteed not to cross a 128-word (page) boundary 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 for uninitialized sections declared with .usect or the .bss directives. The *section names* may optionally be enclosed in quotation marks.

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
```

.sect *Assemble Into Named Section*

Syntax

.sect " *section name* "

Description

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

The *section name* identifies the section. The section name must be enclosed in double quotes. A section name can contain a subsection name in the form *section name* **:** *subsection name*.

See Chapter 2 for more information about sections.

Example

This example defines a special-purpose section named Vars and assembles code into it.

```
 1            ********************************************
 2            **   Begin assembling into .text section.   **
 3            ********************************************
 4 000000                  .text
 5 000000 7600             MOV #120,AC0 ; Assembled into .text
   000002 7808
 6 000004 7B00             ADD #54,AC0 ; Assembled into .text
   000006 3600
 7            ********************************************
 8            **   Begin assembling into Vars section.    **
 9            ********************************************
10 000000                  .sect   "Vars"
11             WORD_LEN    .set    16
12             DWORD_LEN   .set    WORD_LEN * 2
13             BYTE_LEN    .set    WORD_LEN / 2
14 000000 000E             .byte   14
15           ********************************************
16           **   Resume assembling into .text section.  **
17           ********************************************
18 000008                  .text
19 000008 7B00             ADD #66,AC0 ; Assembled into .text
   00000a 4200
20           ********************************************
21           **   Resume assembling into Vars section.   **
22           ********************************************
23 000001                  .sect   "Vars"
24 000001 000D             .field  13, WORD_LEN
25 000002 0A00             .field  0Ah, BYTE_LEN
26 000003 0000             .field  10q, DWORD_LEN
   000004 0008
27
```

.set/.equ — *Define Assembly-Time Constant*

Syntax

symbol **.set** *value*

symbol **.equ** *value*

Description

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

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

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

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

Symbols defined with .set or .equ can be made externally visible with the .def or .global directive (see the .global/.def/.ref topic). In this way, you can define global absolute constants.

Example

This example shows how symbols can be assigned with .set and .equ.

```
 1                ********************************************
 2                **   Set symbol index to an integer expr.   **
 3                **    and use it as an immediate operand.   **
 4                ********************************************
 5                 INDEX   .equ   100/2 +3
 6 000000 7B00            ADD #INDEX,AC0,AC0
   000002 3500
 7
 8                ********************************************
 9                ** Set symbol SYMTAB to a relocatable expr. **
10                **   and use it as a relocatable operand.   **
11                ********************************************
12 000000                  .data
13 000000 000A LABEL       .word  10
14                 SYMTAB  .set   LABEL + 1
15
16                ********************************************
17                **   Set symbol NSYMS equal to the symbol   **
18                **   INDEX and use it as you would INDEX.   **
19                ********************************************
20                 NSYMS   .set   INDEX
21 000001 0035             .word  NSYMS
```

.space *Reserve Space*

Syntax [*label*] **.space** *size in bits*

Description The **.space** directive reserves the number of bits given by *size in bits* in the current section and fill them with 0s. The section program counter is incremented to point to the word following the reserved space.

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

Use the .space Directive in Data Sections

NOTE: Because code and data sections are addressed differently, the use of the .space directive in a section that includes C55x instructions will lead to an invalid access to the data at execution. Consequently, it is highly recommended that this directive be used only in data sections.

Example This example shows how memory is reserved with the .space directive.

```
 1            ******************************************
 2            **  Begin assembling into .data section.   **
 3            ******************************************
 4 000000                .data
 5 000000 0049           .string "In .data"
   000001 006E
   000002 0020
   000003 002E
   000004 0064
   000005 0061
   000006 0074
   000007 0061
 6            ******************************************
 7            **  Reserve 100 bits in the .data section; **
 8            **   RES_1 points to the first word that   **
 9            **         contains reserved bits.         **
10            ******************************************
11 000008      RES_1:   .space 100
12 00000f 000F          .word 15
13 000010 0008"         .word RES_1
14
```

.sslist/.ssnolist *Control Listing of Substitution Symbols*

Syntax

.sslist

.ssnolist

Description

Two directives allow you to control substitution symbol expansion in the listing file:

The **.sslist** directive allows substitution symbol expansion in the listing file. The expanded line appears below the actual source line.

The **.ssnolist** directive suppresses substitution symbol expansion in the listing file.

By default, all substitution symbol expansion in the listing file is suppressed; the assembler acts as if the .ssnolist directive had been used.

Lines with the pound (#) character denote expanded substitution symbols.

Example

This example shows code that, by default, suppresses the listing of substitution symbol expansion, and it shows the .sslist directive assembled, instructing the assembler to list substitution symbol code expansion.

Mnemonic assembly

```
      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                   MOV ADDRA,AC0
      7                   ADD ADDRB,AC0,AC0
      8                   MOV AC0,ADDRB
      9                   .endm
     10
     11 000000C083        MOV AC0,*AR4+
     12 000002            ADD2  ADDRX, ADDRY
1       000002A000-       MOV ADDRX,AC0
1       000004D600        ADD ADDRY,AC0,AC0
        00000600-
1       000007C000-       MOV AC0,ADDRY
     13
     14                   .sslist
     15
     16 000009C083        MOV AC0,*AR4+
     17 00000bC003        MOV AC0,*AR0+
     18
     19 00000d            ADD2  ADDRX, ADDRY
1       00000dA000-       MOV ADDRA,AC0
#                         MOV ADDRX,AC0
1       00000fD600        ADD ADDRB,AC0,AC0
#                         ADD ADDRY,AC0,AC0
        00001100-
1       000012C000-       MOV AC0,ADDRB
#                         MOV AC0,ADDRY
```

Algebraic assembly

```
      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                   AC0 = @(ADDRA)
      7                   AC0 = AC0 + @(ADDRB)
      8                   @(ADDRB) = AC0
      9                   .endm
     10
     11 000000C083        *AR4+ = AC0
     12 000002            ADD2  ADDRX, ADDRY
1       000002A000-       AC0 = @(ADDRX)
1       000004D600        AC0 = AC0 + @(ADDRY)
        00000600-
1       000007C000-       @(ADDRY) = AC0
     13
     14                   .sslist
     15
     16 000009C083        *AR4+ = AC0
     17 00000bC003        *AR0+ = AC0
     18
     19 00000d            ADD2  ADDRX, ADDRY
1       00000dA000-       AC0 = @(ADDRA)
#                         AC0 = @(ADDRX)
1       00000fD600        AC0 = AC0 + @(ADDRB)
#                         AC0 = AC0 + @(ADDRY)
        00001100-
1       000012C000-       @(ADDRB) = AC0
#                         @(ADDRY) = AC0
```

.string/.cstring/.pstring *Initialize Text*

Syntax

.string {*expr*$_1$ | "*string*$_1$"} [**, ... ,** {*expr*$_n$ | "*string*$_n$"}]

.cstring {*expr*$_1$ | "*string*$_1$"} [**, ... ,** {*expr*$_n$ | "*string*$_n$"}]

.pstring {*expr*$_1$ | "*string*$_1$"} [**, ... ,** {*expr*$_n$ | "*string*$_n$"}]

Description

The **.string**, **.cstring**, and **.pstring** directives place 8-bit characters from a character string into the current section. The .string directive places 8-bit characters into consecutive words in the current section. The .pstring directive initializes data in 8-bit chunks, but packs the contents of each string into two characters per word. The *expr* or *string* can be one of the following:

- An expression that the assembler evaluates and treats as an 8- or 16-bit signed number.
- A character string enclosed in double quotes. Each character in a string represents a separate value, and values are stored in consecutive bytes. The entire string *must* be enclosed in quotes.

The .cstring directive adds a NUL character needed by C; the .string directive does not add a NUL character. In addition, .cstring interprets C escapes (\\ \a \b \f \n \r \t \v \<octal>).

Use The .string and .pstring Directives in Data Sections

NOTE: Because code and data sections are addressed differently, the use of the .string and .pstring directives in a section that includes C55x instructions will lead to an invalid access to the data at execution. Consequently, it is highly recommended that these directives be used only in data sections.

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 eight bits. Operands 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.

When you use .string, .cstring, and .pstring in a .struct/.endstruct sequence, the directive only defines a member's size; it does not initialize memory. For more information, see the .struct/.endstruct/.tag topic.

Example

In this example, 8-bit values are placed into consecutive bytes in the current section.

```
1 000000                .data
2 000000 0041           .string  41h, 42h, 43h, 44h
  000001 0042
  000002 0043
  000003 0044
3 000004 0041 Str_Ptr:  .string  "ABCD"
  000005 0042
  000006 0043
  000007 0044
4 000008 4175           .pstring "Austin", "Houston"
  000009 7374
  00000a 696E
  00000b 486F
  00000c 7573
  00000d 746F
  00000e 6E00
5 00000f 0030           .string  36 + 12
```

.struct/.endstruct/.tag *Declare Structure Type*

Syntax

[*stag*]	**.struct**	[*expr*]
[mem_0]	*element*	[$expr_0$]
[mem_1]	*element*	[$expr_1$]
.	.	.
.	.	.
.	.	.
[mem_n]	**.tag** *stag*	[$expr_n$]
.	.	.
.	.	.
.	.	.
[mem_N]	*element*	[$expr_N$]
[*size*]	**.endstruct**	
label	**.tag**	*stag*

Description

The **.struct** directive assigns symbolic offsets to the elements of a data structure definition. This allows you to group similar data elements together and let the assembler calculate the element offset. This is similar to a C structure or a Pascal record. The .struct directive does not allocate memory; it merely creates a symbolic template that can be used repeatedly.

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

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

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

- The *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. A .stag is optional for .struct, but is required for .tag.
- The *expr* is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.
- The $mem_{n/N}$ is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.
- The *element* is one of the following descriptors: .byte, .char, .double, field, .float, .half, .int, .long, .short, .string, .ubyte, .uchar, .uhalt, .uint, .ulong, .ushort, .uword, .word, and .tag. 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. The .tag directive is a special case because stag must be used (as in the definition of stag).
- The $expr_{n/N}$ is an optional expression for the number of elements described. This value defaults to 1. A .string element is considered to be one byte in size, and a .field element is one bit.
- The *size* is an optional label for the total size of the structure.

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

NOTE: The only directives that can appear in a .struct/.endstruct sequence are element descriptors, conditional assembly directives, and the .align directive, which aligns the member offsets on word boundaries. Empty structures are illegal.

The following examples show various uses of the .struct, .tag, and .endstruct directives.

Example 1

```
 1 000000                .data
 2             REAL_REC  .struct               ; stag
 3 0000        NOM       .int                  ; member1 = 0
 4 0001        DEN       .int                  ; member2 = 1
 5 0002        REAL_LEN  .endstruct            ; real_len = 2
 6 000000                .text
 7 000000 D600           ADD @(REAL + REAL_REC.DEN),AC0,AC0
   000002 00-
 8                                             ; access structure element
 9
10 000000                .bss REAL, REAL_LEN    ; allocate mem rec
```

Example 2

```
11                      .data
12             CPLX_REC .struct
13        0000 REALI    .tag REAL_REC          ; stag
14        0002 IMAGI    .tag REAL_REC          ; member1 = 0
15        0004 CPLX_LEN .endstruct             ; cplx_len = 4
16
17             COMPLEX  .tag CPLX_REC          ; assign structure attrib
18
19 000002               .bss COMPLEX, CPLX_LEN
20 000003               .text
21 000003 D600          ADD @(COMPLEX.REALI),AC0,AC0 ; access structure
   000005 00-
22 000006 C000-         MOV AC0,@(COMPLEX.REALI)
23
24 000008 D600          ADD @(COMPLEX.IMAGI),AC1,AC1 ; allocate space
   00000a 11-
```

Example 3

```
 1 000000               .data
 2                      .struct               ; no stag puts mems into
 3                                            ; global symbol table
 4        0000 X        .int                  ; create 3 dim templates
 5        0001 Y        .int
 6        0002 Z        .int
 7        0003          .endstruct
```

Example 4

```
 1 000000               .data
 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 = 66
 7        0043 BIT_LEN  .endstruct            ; length = 67
 8
 9             BITS     .tag    BIT_REC
10 000000               .text
11 000000 D600          ADD @(BITS.BIT7),AC0,AC0 ; move into acc
   000002 00%
12 000003 187F          AND #127,AC0 ; mask off garbage bits
   000005 00
13
14 000000               .bss    BITS, BIT_REC
```

.sst_on/.sst_off *Specify SST Mode*

Syntax

.sst_on

.sst_off

Description

The **.sst_on** and **.sst_off** directives affect the way the assembler encodes certain C54x instructions when ported to C55x. By default, masm55 assumes that the SST bit (saturate on store) is enabled (.sst_on). The default encoding generated by the assembler works whether or not the bit is actually enabled. However, if your code does not enable the SST bit, you may want to use .sst_off, or the −mt assembler option, to allow the assembler to generate a more efficient encoding. For more information, see Section 7.2.1.

The .sst_on directive models the SST status bit set to 1, the default assumption of the assembler. The .sst_off directive models the SST status bit set to 0; this is equivalent to using the -mt assembler option. In the case of a conflict between the command line option and the directive, the directive takes precedence.

The scope of the .sst_on and .sst_off directives is static and not subject to the control flow of the assembly program. All of the assembly code between the .sst_off and the .sst_on directives is assembled with the assumption that SST is disabled.

.symdepend *Effect Symbol Linkage and Visibility*

Syntax

.symdepend *dst symbol name*[**,** *src symbol name*]

Description

The **.symdepend** directive creates an artificial reference from the section defining *src symbol name* to the symbol *dst symbol name*. This prevents the linker from removing the section containing *dst symbol name* if the section defining *src symbol name* is included in the output module. If *src symbol name* is not specified, a reference from the current section is created.

A global *symbol* is defined in the same manner as any other symbol; that is, it appears as a label or is defined by the .set, .equ, .bss, or .usect directive. As with all symbols, if a global symbol is defined more than once, the linker issues a multiple-definition error. The .symdepend directive creates an entry only if the module actually uses the symbol.

If the symbol is *defined in the current module*, the .symdepend directive declares that the symbol and its definition can be used externally by other modules. These types of references are resolved at link time.

.tab *Define Tab Size*

Syntax

.tab *size*

Description

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

Example

In this example, each of the lines of code following a .tab statement consists of a single tab character followed by an NOP instruction.

Source file:

```
; default tab size
    NOP
    NOP
    NOP
      .tab 4
    NOP
    NOP
    NOP
      .tab 16
    NOP
    NOP
    NOP
```

Listing file:

```
      1            ; default tab size
      2 000000 20          NOP
      3 000001 20          NOP
      4 000002 20          NOP
      5
      7 000003 20      NOP
      8 000004 20      NOP
      9 000005 20      NOP
     10
     12 000006 20                  NOP
     13 000007 20                  NOP
     14 000008 20                  NOP
```

.text *Assemble Into the .text Section*

Syntax **.text**

Description The **.text** directive tells the assembler to begin assembling into the .text section, which usually contains executable code. The section program counter is set to 0 if nothing has yet been assembled into the .text section. If code has already been assembled into the .text section, the section program counter is restored to its previous value in the section.

The .text section is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you use a .data or .sect directive to specify a different section.

For more information about sections, see Chapter 2.

Example This example assembles code into the .text and .data sections. The .data section contains integer constants, and the .text section contains executable code.

```
 1            ****************************************
 2            ** Begin assembling into .data section.  **
 3            ****************************************
 4 000000               .data
 5 000000 0041 START:   .string "A","B","C"
   000001 0042
   000002 0043
 6 000003 0058 END:     .string "X","Y","Z"
   000004 0059
   000005 005a
 7            ****************************************
 8            ** Begin assembling into .text section.  **
 9            ****************************************
10 000000               .text
11 000000 D600          ADD START,AC0,AC0
   000002 00"
12 000003 D600          ADD END,AC0,AC0
   000005 00"
13            ****************************************
14            ** Resume assembling into .data section. **
15            ****************************************
16 000006               .data
17 000006 000a          .byte   0Ah, 0Bh
   000007 000b
18 000008 000c          .byte   0Ch, 0Dh
   000009 000d
19            ****************************************
20            ** Resume assembling into .text section. **
21            ****************************************
22 000006               .text
23 000006 2201          MOV AC0,AC1
```

.title — *Define Page Title*

Syntax

.title "*string*"

Description

The **.title** directive supplies a title that is printed in the heading on each listing page. The source statement itself is not printed, but the line counter is incremented.

The *string* is a quote-enclosed title of up to 64 characters. If you supply more than 64 characters, the assembler truncates the string and issues a warning:

```
*** WARNING! line x: W0001: String is too long - will be truncated
```

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

Example

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

Source file:

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

Listing file:

```
TMS320C55x Assembler    Version x.xx        Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Fast Fourier Transforms ****                                  PAGE    1

       2                    ;        .
       3                    ;        .
       4                    ;        .
TMS320C55x Assembler    Version x.xx        Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Floating-Point Routines ****                                  PAGE    2


 No Errors, No Warnings
```

.union/.endunion/.tag *Declare Union Type*

Syntax

[*stag*] **.union** [*expr*]

[mem_0] *element* [$expr_0$]
[mem_1] *element* [$expr_1$]
. . .
. . .
. . .

[mem_n] **.tag** *stag* [$expr_n$]
. . .
. . .
. . .

[mem_N] *element* [$expr_N$]

[*size*] **.endunion**

label **.tag** *stag*

Description

The **.union** directive assigns symbolic offsets to the elements of alternate data structure definitions to be allocated in the same memory space. This enables you to define several alternate structures and then let the assembler calculate the element offset. This is similar to a C union. The .union directive does not allocate any memory; it merely creates a symbolic template that can be used repeatedly.

A .struct definition can contain a .union definition, and .structs and .unions can be nested.

The **.endunion** directive terminates the union definition.

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

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

- The *utag* is the union's tag. is the union's tag. Its value is associated with the beginning of the union. If no utag is present, the assembler puts the union members in the global symbol table with the value of their absolute offset from the top of the union. In this case, each member must have a unique name.
- The *expr* is an optional expression indicating the beginning offset of the union. Unions default to start at 0. This parameter can only be used with a top-level union. It cannot be used when defining a nested union.
- The $mem_{n/N}$ is an optional label for a member of the union. This label is absolute and equates to the present offset from the beginning of the union. A label for a union member cannot be declared global.
- The *element* is one of the following descriptors: .byte, .char, .int, .long, .word, .double, .half, .short, .string, .float, and .field. An element can also be a complete declaration of a nested structure or union, or a structure or union declared by its tag. Following a .union directive, these directives describe the element's size. They do not allocate memory.
- The $expr_{n/N}$ is an optional expression for the number of elements described. This value defaults to 1. A .string element is considered to be one byte in size, and a .field element is one bit.
- The *size* is an optional label for the total size of the union.

Directives That Can Appear in a .union/.endunion Sequence

NOTE: The only directives that can appear in a .union/.endunion sequence are element descriptors, structure and union tags, and conditional assembly directives. Empty structures are illegal.

These examples show unions with and without tags.

Example 1

```
 1                          .global employid
 2 000000                   .data
 3              xample      .union                    ; utag
 4        0000 ival         .word                     ; member1 = 0
 5        0000 fval         .float                    ; member2 = 0
 6        0000 sval         .string                   ; member3 = 0
 7        0002 real_len     .endunion                 ; real_len = 4
 8
 9 000000                   .bss  employid, real_len  ; allocate memory
10
11              employid    .tag  xample
12 000000                   .text
13 000000 D600             ADD @(employid.fval),ADD,ADD ; access union element
   000002 00-
```

Example 2

```
 1 000000                   .data
 2                          .union                    ; utag
 3        0000 x            .long                     ; member1 = long
 4        0000 y            .float                    ; member2 = float
 5        0000 z            .word                     ; member3 = word
 6        0002 size_u       .endunion                 ; real_len = 4
 7
```

.usect *Reserve Uninitialized Space*

Syntax

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

Description

The **.usect** directive reserves space for variables in an uninitialized, named section. This directive is similar to the .bss directive; both simply reserve space for data and that space has no contents. However, .usect defines additional sections that can be placed anywhere in memory, independently of the .bss section.

- The *symbol* points to the first location reserved by this invocation of the .usect directive. The symbol corresponds to the name of the variable for which you are reserving space.
- The *section name* must be enclosed in double quotes. This parameter names the uninitialized section. A section name can contain a subsection name in the form *section name* **:** *subsection name*.
- The *size in bytes* is an expression that defines the number of bytes that are reserved in *section name*.
- The *blocking flag* is an optional parameter that, if specified and nonzero, means 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.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. This boundary indicates the size of the slot in bytes and can be set to any power of 2.

Specifying an Alignment Flag Only

NOTE: To specify an alignment flag without a blocking flag, you must insert two commas before the alignment flag, as shown in the syntax.

Initialized sections directives (.text, .data, and .sect) end the current section and tell the assembler to begin assembling into another section. A .usect or .bss directive encountered in the current section is simply assembled, and assembly continues in the current section.

Variables that can be located contiguously in memory can be defined in the same specified section; to do so, repeat the .usect directive with the same section name and the subsequent symbol (variable name).

For more information about sections, see Chapter 2.

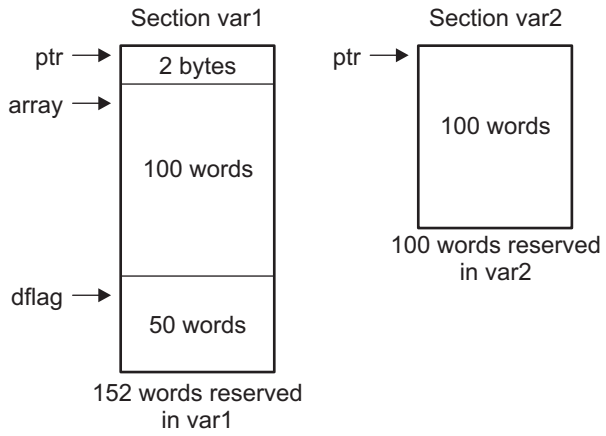
Example

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 bytes reserved in var1, and dflag points to the first word in a block of 50 bytes in var1. The symbol vec points to the first word reserved in the var2 section.

Figure 4-8 shows how this example reserves space in two uninitialized sections, var1 and var2.

```
 1              ****************************************
 2              **     Assemble into .text section.     **
 3              ****************************************
 4 000000                .text
 5 000000 3C30          MOV #3,AC0
 6
 7              ****************************************
 8              **       Reserve 1 word in var1.        **
 9              ****************************************
10 000000       ptr     .usect "var1", 1
11
12              ****************************************
13              **      Reserve 100 words in var1.      **
14              ****************************************
15 000001       array   .usect "var1", 100
16
17 000002 7B00          ADD #55,AC0,AC0   ; Still in .text
   000004 3700
18
19              ****************************************
20              **      Reserve 50 words in var1.       **
21              ****************************************
22 000065       dflag   .usect "var1", 50
23
24 000006 7B06         ADD #dflag,AC0,AC0 ; Still in .text
   000008 5000-
25
26              ****************************************
27              **      Reserve 100 words in var2.      **
28              ****************************************
29 000000       vec     .usect "var2", 100
30
31 00000a 7B00          ADD #vec,AC0,AC0  ; Still in .text
   00000c 0000-
32              ****************************************
33              **  Declare an external .usect symbol.  **
34              ****************************************
35                      .global array
```

Figure 4-8. The .usect Directive



.unasg/.undefine *Turn Off Substitution Symbol*

Syntax

.unasg *symbol*

.undefine *symbol*

Description

The **.unasg** and **.undefine** directives remove the definition of a substitution symbol created using .asg or .define. The named *symbol* will removed from the substitution symbol table from the point of the .undefine or .unasg to the end of the assembly file. See Section 3.9.6 for more information on substitution symbols.

These directives can be used to remove from the assembly environment any C/C++ macros that may cause a problem. See Chapter 14 for more information about using C/C++ headers in assembly source.

.var *Use Substitution Symbols as Local Variables*

Syntax

.var sym_1 [**,** sym_2 **,** ... **,** sym_n]

Description

The **.var** directive allows you to use substitution symbols as local variables within a macro. With this directive, you can define up to 32 local macro substitution symbols (including parameters) per macro.

The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

See Section 3.9.6 for more information on substitution symbols .See Chapter 5 for information on macros.

.vli_on/.vli_off *Suppress Variable-Length Instruction Resolution*

Syntax

.vli_on

.vli_off

Description

The **.vli_on** and **.vli_off** directives affect the way the assembler handles variable-length instructions. The .vli_off directive is equivalent to using the −atv command line option. In the case of a conflict between the command line option and the directive, the directive takes precedence.

By default (.vli_on), the assembler attempts to resolve all stand-alone, variable-length instructions to their smallest possible size.

Size resolution is performed on the following instruction groups:

```
goto L7, L16, P24
if (cond) goto l4
if (cond) goto L8, L16, P24
call L16, P24
if (cond) call L16, P24
```

In some cases, you may want the assembler to keep the largest (P24) form of certain instructions. The P24 versions of certain variable-length instructions execute in fewer cycles than the smaller version of the same instructions. Use the .vli_off directive to keep the following instructions in their largest form:

```
goto P24
call P24
```

The .vli_off and .vli_on directives can be used to toggle this behavior for regions of an assembly file. All other variable-length instructions will continue to be resolved to their smallest possible size by the assembler, despite the use of the .vli_off directive.

The scope of the .vli_off and .vli_on directives is static and not subject to the control flow of the assembly program.

.warn_on/.warn_off *Control Assembler Warning Messages*

Syntax

.warn_on

.warn_off

Description

The **.warn_on** and **.warn_off** directives control the reporting of assembler warning messages. By default (.warn_on), the assembler will generate warning messages. The .warn_off directive suppresses assembler warning messages and is equivalent to using the -mw command line option. In the case of a conflict between the command line option and the directive, the directive takes precedence.

The .warn_off and .warn_on directives can be used to toggle this behavior for regions of an assembly file.

The scope of the .warn_off and .warn_on directives is static and not subject to the control flow of the assembly program. Warnings will not be reported for any assembly code between the .warn_off and .warn_on directives within a file.

Macro Description

The TMS320C55x 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**

5.1 Using Macros

Programs often contain routines that are executed several times. Instead of repeating the source statements for a routine, you can define the routine as a macro, then call the macro in the places where you would normally repeat the routine. This simplifies and shortens your source program.

If you want to call a macro several times but with different data each time, you can assign parameters within a macro. This enables you to pass different information to the macro each time you call it. The macro language supports a special symbol called a *substitution symbol*, which is used for macro parameters. See Section 5.3 for more information.

Using a macro is a 3-step process.

Step 1. **Define the macro.** You must define macros before you can use them in your program. There are two methods for defining macros:

(a) Macros can be defined at the beginning of a *source file* or in a copy/include file. See Section 5.2, *Defining Macros*, for more information.

(b) Macros can also be defined in a *macro library*. A macro library is a collection of files in archive format created by the archiver. Each member of the archive file (macro library) may contain one macro definition corresponding to the member name. You can access a macro library by using the .mlib directive. For more information, see Section 5.4.

Step 2. **Call the macro.** After you have defined a macro, call it by using the macro name as a mnemonic in the source program. This is referred to as a *macro call*.

Step 3. **Expand the macro.** The assembler expands your macros when the source program calls them. During expansion, the assembler passes arguments by variable to the macro parameters, replaces the macro call statement with the macro definition, then assembles the source code. By default, the macro expansions are printed in the listing file. You can turn off expansion listing by using the .mnolist directive. For more information, see Section 5.8.

When the assembler encounters a macro definition, it places the macro name in the opcode table. This redefines any previously defined macro, library entry, directive, or instruction mnemonic that has the same name as the macro. This allows you to expand the functions of directives and instructions, as well as to add new instructions.

5.2 Defining Macros

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file or in a .copy/.include file (see Copy Source File); they can also be defined in a macro library. For more information about macro libraries, see Section 5.4.

Macro definitions can be nested, and they can call other macros, but all elements of the macro must be defined in the same file. Nested macros are discussed in Section 5.9.

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

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

macname	names the macro. You must place the name in the source statement's label field. Only the first 128 characters of a macro name are significant. The assembler places the macro name in the internal opcode table, replacing any instruction or previous macro definition with the same name.
.macro	is the directive that identifies the source statement as the first line of a macro definition. You must place .macro in the opcode field.
parameter $_1$, *parameter* $_n$	are optional substitution symbols that appear as operands for the .macro directive. Parameters are discussed in Section 5.3.

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	is a directive that functions as a *goto .endm*. The .mexit directive is useful when error testing confirms that macro expansion fails and completing the rest of the macro is unnecessary.
.endm	is the directive that terminates the macro definition.

If you want to include comments with your macro definition but *do not* want those comments to appear in the macro expansion, use an exclamation point to precede your comments. If you *do* want your comments to appear in the macro expansion, use an asterisk or semicolon. See Section 5.7 for more information about macro comments.

Example 5-1 and Example 5-2 show the definition, call, and expansion of a macro.

Example 5-1. Mnemonic Macro Definition, Call, and Expansion

```
      1              *
      2
      3              *        add3
      4              *
      5              *        ADDRP = P1 + P2 + P3
      6
      7              add3     .macro P1, P2, P3, ADDRP
      8
      9                       MOV P1,AC0
     10                       ADD P2,AC0,AC0
     11                       ADD P3,AC0,AC0
     12                       MOV AC0,ADDRP
     13                       .endm
     14
     15
     16                       .global abc, def, ghi, adr
     17
     18 000000               add3 abc, def, ghi, adr
1
1       000000 A000!          MOV abc,AC0
1       000002 D600           ADD def,AC0,AC0
        000004 00!
1       000005 D600           ADD ghi,AC0,AC0
        000007 00!
1       000008 C000!          MOV AC0,adr
```

Example 5-2. Algebraic Macro Definition, Call, and Expansion

```
      1              *
      2
      3              *        add3
      4              *
      5              *        ADDRP = P1 + P2 + P3
      6
      7              add3     .macro P1, P2, P3, ADDRP
      8
      9                       AC0 = @(P1)
     10                       AC0 = AC0 + @(P2)
     11                       AC0 = AC0 + @(P3)
     12                       @(ADDRP) = AC0
     13                       .endm
     14
     15
     16                       .global abc, def, ghi, adr
     17
     18 000000               add3 abc, def, ghi, adr
1
1       000000 A000!          AC0 = @(abc)
1       000002 D600           AC0 = AC0 + @(def)
        000004 00!
1       000005 D600           AC0 = AC0 + @(ghi)
        000007 00!
1       000008 C000!          MOV AC0,adr@(adr) = AC0
```

5.3 Macro Parameters/Substitution Symbols

If you want to call a macro several times with different data each time, you can assign parameters within the macro. The macro language supports a special symbol, called a *substitution symbol*, which is used for macro parameters.

Macro parameters are substitution symbols that represent a character string. These symbols can also be used outside of macros to equate a character string to a symbol name (see Section 3.9.6).

Valid substitution symbols can be up to 128 characters long and *must begin with a letter*. The remainder of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

Substitution symbols used as macro parameters are local to the macro they are defined in. You can define up to 32 local substitution symbols (including substitution symbols defined with the .var directive) per macro. For more information about the .var directive, see Section 5.3.6.

During macro expansion, the assembler passes arguments by variable to the macro parameters. The character-string equivalent of each argument is assigned to the corresponding parameter. Parameters without corresponding arguments are set to the null string. If the number of arguments exceeds the number of parameters, the last parameter is assigned the character-string equivalent of all remaining arguments.

If you pass a list of arguments to one parameter or if you pass a comma or semicolon to a parameter, you must surround these terms with quotation marks.

At assembly time, the assembler replaces the macro parameter/substitution symbol with its corresponding character string, then translates the source code into object code.

Example 5-3 shows the expansion of a macro with varying numbers of arguments.

Example 5-3. 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
```

5.3.1 Directives That Define Substitution Symbols

You can manipulate substitution symbols with the **.asg** and **.eval** directives.

- The **.asg** directive assigns a character string to a substitution symbol.

  For the .asg directive, the quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the *substitution symbol*. The syntax of the .asg directive is:

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

Example 5-4 shows character strings being assigned to substitution symbols.

Example 5-4. The .asg Directive

```
.asg   AR0,FP            ; frame pointer
```

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

  The .eval directive evaluates the *expression* and assigns the string value of the result to the *substitution symbol*. If the expression is not well defined, the assembler generates an error and assigns the null string to the symbol. The syntax of the .eval directive is:

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

Example 5-5 shows arithmetic being performed on substitution symbols.

Example 5-5. The .eval Directive

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

In Example 5-5, the .asg directive could be replaced with the .eval directive (.eval 1, counter) without changing the output. In simple cases like this, you can use .eval and .asg interchangeably. However, you must use .eval if you want to calculate a *value* from an expression. While .asg only assigns a character string to a substitution symbol, .eval evaluates an expression and then assigns the character string equivalent to a substitution symbol.

See Assign a Substitution Symbol for more information about the .asg and .eval assembler directives.

5.3.2 Built-In Substitution Symbol Functions

The following built-in substitution symbol functions enable you to make decisions on the basis of the string value of substitution symbols. These functions always return a value, and they can be used in expressions. Built-in substitution symbol functions are especially useful in conditional assembly expressions. Parameters of these functions are substitution symbols or character-string constants.

In the function definitions shown in Table 5-1, *a* and *b* are parameters that represent substitution symbols or character-string constants. The term *string* refers to the string value of the parameter. The symbol *ch* represents a character constant.

Table 5-1. Substitution Symbol Functions and Return Values

Function	Return Value
$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)	1 if string *a* is a valid predefined register name
	0 if string *a* is not a valid predefined register name

(1) For more information about predefined register names, see Section 3.9.5.

Example 5-6 shows built-in substitution symbol functions.

Example 5-6. Using Built-In Substitution Symbol Functions

```
.asg  label, ADDR                ; ADDR = label
.if   ($symcmp(ADDR,"label") = 0) ; evaluates to true
SUB ADDR,AC0,AC0
.endif
.asg  "x,y,z" , list             ; list = x,y,z
.if   ($ismember(ADDR,list))     ; addr = x, list = y,z
SUB ADDR,AC0,AC0 ; sub x
.endif
```

5.3.3 Recursive Substitution Symbols

When the assembler encounters a substitution symbol, it attempts to substitute the corresponding character string. If that string is also a substitution symbol, the assembler performs substitution again. The assembler continues doing this until it encounters a token that is not a substitution symbol or until it encounters a substitution symbol that it has already encountered during this evaluation.

In Example 5-7, the x is substituted for z; z is substituted for y; and y is substituted for x. The assembler recognizes this as infinite recursion and ceases substitution.

Example 5-7. 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,AC0,AC0  ; recursive expansion
```

5.3.4 Forced Substitution

In some cases, substitution symbols are not recognizable to the assembler. The forced substitution operator, which is a set of colons surrounding the symbol, enables you to force the substitution of a symbol's character string. Simply enclose a symbol with colons to force the substitution. Do not include any spaces between the colons and the symbol.

The syntax for the forced substitution operator is:

:symbol:

The assembler expands substitution symbols surrounded by colons before expanding other substitution symbols.

You can use the forced substitution operator only inside macros, and you cannot nest a forced substitution operator within another forced substitution operator.

Example 5-8 shows how the forced substitution operator is used.

Example 5-8. Using the Forced Substitution Operator

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

The force macro would generate the following source code:

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

5.3.5 Accessing Individual Characters of Subscripted Substitution Symbols

In a macro, you can access the individual characters (substrings) of a substitution symbol with subscripted substitution symbols. You must use the forced substitution operator for clarity.

You can access substrings in two ways:

- :symbol (*well-defined expression*):

  This method of subscripting evaluates to a character string with one character.

- :symbol (*well-defined 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 5-9 and Example 5-10 show built-in substitution symbol functions used with subscripted substitution symbols.

In Example 5-9, subscripted substitution symbols redefine the ADD instruction so that it handles short immediates. In Example 5-10, 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.

Example 5-9. Using Subscripted Substitution Symbols to Redefine an Instruction

```
ADDX       .macro   ABC
           .var     TMP
           .asg     :ABC(1):,TMP
           .if      $symcmp(TMP,"#") = 0
           ADD ABC,AC0,AC0
           .else
           .emsg    "Bad Macro Parameter"
           .endif
           .endm
           ADDX     #100           ;macro call
           ADDX     *AR1           ;macro call
```

Example 5-10. 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
          .data
          .word pos
```

5.3.6 Substitution Symbols as Local Variables in Macros

If you want to use substitution symbols as local variables within a macro, you can use the **.var** directive to define up to 32 local macro substitution symbols (including parameters) per macro. The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

.var sym_1 [,sym_2 , ... ,sym_n]

The .var directive is used in Example 5-9 and Example 5-10.

5.4 Macro Libraries

One way to define macros is by creating a macro library. A macro library is a collection of files that contain macro definitions. You must use the archiver to collect these files, or members, into a single file (called an archive). Each member of a macro library contains one macro definition. The files in a macro library must be unassembled source files. The macro name and the member name must be the same, and the macro filename's extension must be .asm. For example:

Macro Name	Filename in Macro Library
simple	simple.asm
add3	add3.asm

You can access the macro library by using the .mlib assembler directive (described in Define Macro Library). The syntax is:

.mlib *filename*

When the assembler encounters the .mlib directive, it opens the library named by filename and creates a table of the library's contents. The assembler enters the names of the individual members within the library into the opcode tables as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table.

The assembler expands the library entry in the same way it expands other macros. See Section 5.1 for how the assembler expands macros. You can control the listing of library entry expansions with the .mlist directive. For more information about the .mlist directive, see Section 5.8 and Start/Stop Macro Expansion Listing. Only macros that are actually called from the library are extracted, and they are extracted only once.

You can use the archiver to create a macro library by including the desired files in an archive. A macro library is no different from any other archive, except that the assembler expects the macro library to contain macro definitions. The assembler expects *only* macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable results. For information about creating a macro library archive, see Section 8.1.

5.5 Using Conditional Assembly in Macros

The conditional assembly directives are **.if/.elseif/.else/.endif** and **.loop/ .break/.endloop**. They can be nested within each other up to 32 levels deep. The format of a conditional block is:

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

The **.elseif** and **.else** directives are optional in conditional assembly. The .elseif directive can be used more than once within a conditional assembly code block. When .elseif and .else are omitted and when the .if expression is false (0), the assembler continues to the code following the .endif directive. See Assemble Conditional Blocks for more information on the .if/ .elseif/.else/.endif directives.

The **.loop/.break/.endloop** directives enable you to assemble a code block repeatedly. The format of a repeatable block is:

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

The **.loop** directive's optional *well-defined expression* evaluates to the loop count (the number of loops to be performed). If the expression is omitted, the loop count defaults to 1024 unless the assembler encounters a .break directive with an expression that is true (nonzero). See Assemble Conditional Blocks Repeatedly for more information on the .loop/.break/.endloop directives.

The **.break** directive and its expression are optional in repetitive assembly. If the expression evaluates to false, the loop continues. The assembler breaks the loop when the .break expression evaluates to true or when the .break expression is omitted. When the loop is broken, the assembler continues with the code after the .endloop directive.

For more information, see Section 4.7.

Example 5-11, Example 5-12, and Example 5-13 show the .loop/.break/ .endloop directives, properly nested conditional assembly directives, and built-in substitution symbol functions used in a conditional assembly code block.

Example 5-11. The .loop/.break/.endloop Directives

```
.asg    1,x
.loop

.break  (x == 10)  ; if x == 10, quit loop/break with expression

.eval   x+1,x
.endloop
```

Example 5-12. Nested Conditional Assembly Directives

```
      .asg    1,x
      .loop

      .if     (x == 10)  ; if x == 10, quit loop
      .break  (x == 10)  ; force break
      .endif

      .eval   x+1,x
      .endloop
```

Example 5-13. Built-In Substitution Symbol Functions in a Conditional Assembly Code Block

```
      .ref OPZ
      .fcnolist
*
*Double Add or Subtract
*
DB    .macro ABC, ADDR, dst ; add or subtract double

      .if  $symcmp(ABC,"+") == 0
      ADD dbl(ADDR),dst ; add double

      .elseif $symcmp(ABC,"-") == 0
      SUB dbl(ADDR),dst ; subtract double

      .else
      .emsg "Incorrect Operator Parameter"

.endif

.endm

*Macro Call
      DB   -, @OPZ, AC0
```

5.6 Using Labels in Macros

All labels in an assembly language program must be unique. This includes labels in macros. If a macro is expanded more than once, its labels are defined more than once. *Defining a label more than once is illegal*. The macro language provides a method of defining labels in macros so that the labels are unique. Simply follow each label with a question mark, and the assembler replaces the question mark with a period followed by a unique number. When the macro is expanded, *you do not see the unique number in the listing file*. Your label appears with the question mark as it did in the macro definition. You cannot declare this label as global. The syntax for a unique label is:

label **?**

Example 5-14 and Example 5-15 show unique label generation in a macro. The maximum label length is shortened to allow for the unique suffix. For example, if the macro is expanded fewer than 10 times, the maximum label length is 126 characters. If the macro is expanded from 10 to 99 times, the maximum label length is 125. The label with its unique suffix is shown in the cross-listing file. To obtain a cross-listing file, invoke the assembler with the --cross_reference option (see Section 3.3).

Example 5-14. Unique Labels in a Macro in Mnemonic Assembly

```
      1              ; define macro
      2              MLAB     .macro AVAR, BVAR ; find minimum
      3
      4                       MOV AVAR,AC0
      5                       SUB #BVAR,AC0,AC0
      6                       BCC M1?,AC0 < #0
      7                       MOV #BVAR,AC0
      8                       B M2?
      9              M1?      MOV AVAR,AC0
     10              M2?
     11                      .endm
     12
     13              ; call macro
     14 000000               MLAB 50, 100
1
1       000000 A064           MOV 50,AC0
1       000002 7C00           SUB #100,AC0,AC0
        000004 6400
1       000006 6320           BCC M1?,AC0 < #0
1       000008 7600           MOV #100,AC0
        00000a 6408
1       00000c 4A02           B M2?
1       00000e A064   M1?     MOV 50,AC0
1       000010        M2?
```

Example 5-15. Unique Labels in a Macro in Algebraic Assembly

```
      1              ; define macro
      2              MLAB     .macro AVAR, BVAR ; find minimum
      3
      4                       AC0 = @(AVAR)
      5                       AC0 = AC0 - #(BVAR)
      6                       if (AC0 < #0) goto #(M1?)
      7                       AC0 = #(BVAR)
      8                       goto #(M2?)
      9              M1?      AC0 = @(AVAR)
     10              M2?
     11                       .endm
     12
     13              ; call macro
     14 000000                MLAB 50, 100
1
1       000000 A064           AC0 = @(50)
1       000002 7C00           AC0 = AC0 - #(100)
        000004 6400
1       000006 6320           if (AC0 < #0) goto #(M1?)
1       000008 7600           AC0 = #(100)
        00000a 6408
1       00000c 4A02           goto #(M2?)
1       00000e A064  M1?      AC0 = @(50)
1       000010       M2?
```

5.7 Producing Messages in Macros

The macro language supports three directives that enable you to define your own assembly-time error and warning messages. These directives are especially useful when you want to create messages specific to your needs. The last line of the listing file shows the error and warning counts. These counts alert you to problems in your code and are especially useful during debugging.

.emsg sends error messages to the listing file. The .emsg directive generates errors in the same manner as the assembler, incrementing the error count and preventing the assembler from producing an object file.

.mmsg sends assembly-time messages to the listing file. The .mmsg directive functions in the same manner as the .emsg directive but does not set the error count or prevent the creation of an object file.

.wmsg sends warning messages to the listing file. The .wmsg directive functions in the same manner as the .emsg directive, but it increments the warning count and does not prevent the generation of an object file.

Macro comments are comments that appear in the definition of the macro *but do not show up in the expansion of the macro*. An exclamation point in column 1 identifies a macro comment. If you want your comments to appear in the macro expansion, precede your comment with an asterisk or semicolon.

Example 5-16 shows user messages in macros and macro comments that do not appear in the macro expansion.

For more information about the .emsg, .mmsg, and .wmsg assembler directives, see Define Messages.

Example 5-16. Producing Messages in a Macro

```
      1             testparam  .macro x,y
      2
      2                        .if ($symlen(x) == 0)
      3                        .emsg "ERROR -- Missing Parameter"
      4                        .mexit
      5                        .elseif ($symlen(y) == 0)
      6                        .emsg "ERROR == Missing Parameter"
      7                        .mexit
      8                        .else
      9                        MOV y,AC0
     10                        MOV x,AC0
     11                        ADD AC0,AC1
     12                        .endif
     13                        .endm
     14
     15 000000                testparam 1,2
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       000000 A004            MOV 2,AC0
1       000002 A102            MOV 1,AC1
1       000004 2401            ADD AC0,AC1
1                              .endif
     17 000006                testparam
1                              .if ($symlen(x) == 0)
1                              .emsg "ERROR -- Missing Parameter"
  ***** USER ERROR ***** - : ERROR -- Missing Parameter
1                               .mexit

 1 Error, No Warnings
```

5.8 Using Directives to Format the Output Listing

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

By default, the assembler shows macro expansions and false conditional blocks in the list output file. You may want to turn this listing off or on within your listing file. Four sets of directives enable you to control the listing of this information:

- **Macro and loop expansion listing**
  - **.mlist** expands macros and .loop/.endloop blocks. The .mlist directive prints all code encountered in those blocks.
  - **.mnolist** suppresses the listing of macro expansions and .loop/ .endloop blocks.

  For macro and loop expansion listing, .mlist is the default.
- **False conditional block listing**
  - **.fclist** causes the assembler to include in the listing file all conditional blocks that do not generate code (false conditional blocks). Conditional blocks appear in the listing exactly as they appear in the source code.
  - **.fcnolist** suppresses the listing of false conditional blocks. Only the code in conditional blocks that actually assemble appears in the listing. The .if, .elseif, .else, and .endif directives do not appear in the listing.

  For false conditional block listing, .fclist is the default.
- **Substitution symbol expansion listing**
  - **.sslist** expands substitution symbols in the listing. This is useful for debugging the expansion of substitution symbols. The expanded line appears below the actual source line.
  - **.ssnolist** turns off substitution symbol expansion in the listing.

  For substitution symbol expansion listing, .ssnolist is the default.
- **Directive listing**
  - **.drlist** causes the assembler to print to the listing file all directive lines.
  - **.drnolist** suppresses the printing of certain directives in the listing file. These directives are .asg, .eval, .var, .sslist, .mlist, .fclist, .ssnolist, .mnolist, .fcnolist, .emsg, .wmsg, .mmsg, .length, .width, and .break.

  For directive listing, .drlist is the default.

5.9 Using Recursive and Nested Macros

The macro language supports recursive and nested macro calls. This means that you can call other macros in a macro definition. You can nest macros up to 32 levels deep. When you use recursive macros, you call a macro from its own definition (the macro calls itself).

When you create recursive or nested macros, you should pay close attention to the arguments that you pass to macro parameters because the assembler uses dynamic scoping for parameters. This means that the called macro uses the environment of the macro from which it was called.

Example 5-17 shows nested macros. The y in the in_block macro hides the y in the out_block macro. The x and z from the out_block macro, however, are accessible to the in_block macro.

Example 5-17. Using Nested Macros

```
in_block  .macro y,a
             .          ; visible parameters are y,a and x,z from the calling macro
          .endm

out_block .macro    x,y,z
             .          ; visible parameters are x,y,z
             .
          in_block x,y  ; macro call with x and y as arguments
             .
             .
          .endm
          out_block     ; macro call
```

Example 5-18 and Example 5-19 show recursive and fact macros. The fact macro produces assembly code necessary to calculate the factorial of n, where n is an immediate value. The result is placed in data memory address loc . The fact macro accomplishes this by calling fact1, which calls itself recursively.

Example 5-18. Using Recursive Macros in Mnemonic Assembly

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

       MOV #1,loc
       .else
       MOV #N,loc      ; 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
       MOV loc,T3      ; multiply present factorial
       MOV T3,HI(AC2)  ; by present position
       MPYK #N,AC2,AC0
       MOV AC0,loc     ; save result
       .eval N - 1, N  ; decrement position
       fact1           ; recursive call
       .endif

       .endm
```

Example 5-19. Using Recursive Macros in Algebraic Assembly

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

       loc = #1
       .else
       loc = #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
       T3 = loc        ; multiply present factorial
       HI(AC2) = T3    ; by present position
       AC0 = AC2 * #(N)
       loc = AC0       ; save result
       .eval N - 1, N  ; decrement position
       fact1           ; recursive call
       .endif

       .endm
```

5.10 Macro Directives Summary

The directives listed in Table 5-2 through Table 5-6 can be used with macros. The .macro, .mexit, .endm and .var directives are valid only with macros; the remaining directives are general assembly language directives.

Table 5-2. Creating Macros

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.endm	End macro definition	Section 5.2	.endm
macname **.macro** [*parameter*$_1$][,... , *parameter*$_n$]	Define macro by *macname*	Section 5.2	.macro
.mexit	Go to .endm	Section 5.2	Section 5.2
.mlib *filename*	Identify library containing macro definitions	Section 5.4	.mlib

Table 5-3. Manipulating Substitution Symbols

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.asg ["]*character string*["], *substitution symbol*	Assign character string to substitution symbol	Section 5.3.1	.asg
.eval *well-defined expression, substitution symbol*	Perform arithmetic on numeric substitution symbols	Section 5.3.1	.eval
.var *sym*$_1$ [, *sym*$_2$, ..., *sym*$_n$]	Define local macro symbols	Section 5.3.6	.var

Table 5-4. Conditional Assembly

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.break [*well-defined expression*]	Optional repeatable block assembly	Section 5.5	.break
.endif	End conditional assembly	Section 5.5	.endif
.endloop	End repeatable block assembly	Section 5.5	.endloop
.else	Optional conditional assembly block	Section 5.5	.else
.elseif *well-defined expression*	Optional conditional assembly block	Section 5.5	.elseif
.if *well-defined expression*	Begin conditional assembly	Section 5.5	.if
.loop [*well-defined expression*]	Begin repeatable block assembly	Section 5.5	.loop

Table 5-5. Producing Assembly-Time Messages

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.emsg	Send error message to standard output	Section 5.7	.emsg
.mmsg	Send assembly-time message to standard output	Section 5.7	.mmsg
.wmsg	Send warning message to standard output	Section 5.7	.wmsg

Table 5-6. Formatting the Listing

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.fclist	Allow false conditional code block listing (default)	Section 5.8	.fclist
.fcnolist	Suppress false conditional code block listing	Section 5.8	.fcnolist
.mlist	Allow macro listings (default)	Section 5.8	.mlist
.mnolist	Suppress macro listings	Section 5.8	.mnolist
.sslist	Allow expanded substitution symbol listing	Section 5.8	.sslist
.ssnolist	Suppress expanded substitution symbol listing (default)	Section 5.8	.ssnolist

Running C54x Code on C55x

In addition to accepting TMS320C55x source code, the C55x mnemonic assembler also accepts TMS320C54x™ mnemonic assembly. The C54x instruction set contains 211 instructions; the C55x mnemonic instruction set is a superset of the C54x instruction set. The table below contains statistics on how the C54x instructions assemble with masm55:

Original C54x instruction assembles as:	% of total C54x instruction set	% of commonly used C54x instructions
One C55x instruction	85	95-99
Two C55x instructions	10	1-3
More than two C55x instructions	5	0-2

The data in the second column characterizes the assembly of an imaginary file containing an instance of every C54x instruction. However, the instructions that assemble as more than two instructions are not commonly used. The data in the third column characterizes the assembly of a file containing the most commonly used C54x instructions. Exact percentages depend on the specific source file used.

Because of this compatibility, the C55x mnemonic assembler can assemble C54x code to generate C55x object code, that upon execution, computes exactly the same result. This assembler feature preserves your C54x source code investment as you transition to the C55x.

This chapter does not explain how to take advantage of the new architecture features of the C55x. For this type of information, see the *TMS320C55x DSP Programmer's Guide*.

Topic **Page**

6.1 C54x to C55x Development Flow

To run a C54x application on the C55x, you must:

- Assemble each function with cl55. Your C54x application should already assemble without errors with the cl500 assembler. For information on cl55 options that support the porting of C54x code, see Section 7.2.
- Initialize the stack pointers SP and SSP. See Section 6.1.1.
- Handle differences in memory placement. See Section 6.1.2.
- Update your C54x linker command file for C55x. See Section 6.1.3.

To use ported C54x functions along with native C55x functions, see Section 7.3.

6.1.1 Initializing the Stack Pointers

When you execute ported C54x code from reset, the appropriate run-time environment is already in place. However, it is still necessary to initialize the stack pointers SP (primary stack) and SSP (secondary system stack). For example:

```
stack_size   .set 0x400
stack:       .usect "stack_section", stack_size
sysstack:    .usect "stack_section", stack_size
             AMOV #(stack+stack_size), XSP
             MOV #(sysstack+stack_size), SSP
```

The stacks grow from high addresses to low addresses, so the stack pointers must be initialized to the highest address. The primary stack and the secondary system stack must be within the same 64K word page of memory.

Code that modifies the SP can be ported. Such modification can be done directly or indirectly. In some case you will receive not receive warnings that the SSP must also be modified.

6.1.2 Handling Differences in Memory Placement

This section describes the limitations on where you can place your code in memory. All data must be placed in the first 64K words.

If your C54x code includes any of the following, all code must be placed in the first 64K bytes:

- Indirect calls with CALA
- Modification of the repeat block address registers REA or RSA
- Indirect branches with BACC, if you do not use the -v option for specifying the device revision.

If your C54x code includes either of the following, it can be placed in any 64K byte block without crossing the 64K byte boundary:

- Indirect branches with BACC, provided you build with the appropriate the −v option for specifying the device revision.
- Modification or use of the function return address on the stack in a non-standard way (stack unwinding)

Otherwise, code can be placed anywhere in memory.

6.1.3 Updating a C54x Linker Command File

You must take the following information into consideration when updating a C54x linker command file for use in a C55x system.

- In a C55x linker command file, all addresses and lengths (for both code and data) are expressed in bytes. Data is expressed in bytes even though it is addressed in words on the processor. Consequently, the− heap and −stack options specify the bytes, not words, to be allocated.
- On C54x, memory is split into two different pages: page 0 for code and page 1 for data. The address space on each page ranges from 0 to 0xFFFF (in words). The C55x has a single, unified address space ranging from 0 to 0xFFFFFF.
- On C55x, all sections must have a unique address, and may not overlap. On C54x, where code and data are on different pages, sections can have the same address, and they can overlap.

- If you use DP-based direct memory addressing (DMA), be sure that you don't change the relationship between the DP boundaries and variables accessed with DMA. On C54x, DP pages are 128 words long and must begin on 128-word boundaries. C54x code ported by cl55 must adhere to the same restriction. However, the restriction is expressed differently in the linker command file. Because the linker uses byte addresses, a DP page is 256 bytes long and must begin on a 256-byte boundary.

  You can place variables on the same DP page by using the blocking parameter of the .bss or .usect assembler directive. If you use the blocking parameter, you don't need to modify your linker command file.

  To use the linker command file to arrange variables on the same DP page, you must change a specification of 128 words to be 256 bytes. For example, you must change a specification such as:

  output_section ALIGN(128) { *list of input sections* }

  to be:

  output_section ALIGN(256) { *list of input sections* }

6.2 Understanding the Listing File

The assembler's listing file (created when invoking cl55 with the --asm_listing option) provides additional information on how C54x instructions are mapped for the C55x.

Consider the following example C54x source file:

```
       .global name

       ADD     *AR2, A
       LD      *AR3, B

       RPT     #10
       MVDK    *AR4+, name

subm   .macro  mem1, mem2, reg
       LD      mem1, reg
       SUB     mem2, reg
       .endm

       subm    name, *AR6, B

       MOV     T1, AC3 ; native C55x instruction
```

The listing file shown below has explanations inserted for clarification.

The file begins with a comment on a C55x temporary register used in porting the file.

```
16         ; Temporary Registers Used: XCDP
```

This comment appears only when temporary registers are necessary in the porting of the code. The temporary registers are used in the encodings that begin with a !REG! comment later in the file (as shown in line 7 of this example).

```
1                  .global name
2
3 000000 D641      ADD *AR2,A
  000002 00
```

Notice A in the example above is accepted even though it maps to AC0 on the C55x.

C54x instructions with a different syntax in C55x but a single-line mapping also appear without any special notation:

```
4 000003 A161      LD *AR3, B
```

The LD instruction above could be written as:

```
                   MOV *AR3, AC1
```

The code below shows a multiple-line instruction mapping that requires the C55x instructions to be in a different order than the original source. Because this multiple-line encoding requires the use of a C55x temporary register, it starts with a !REG! line that echoes the original source. The multiple lines that correspond to the mapping will begin and end with the original source line number (7, in this case).

```
7 ****** !REG!      MVDK *AR4+, name
7 000005 EC31       AMAR *(#(name)), XCDP ; port of
  000007 7E00                             ; MVDK *AR4+, name
  000009 0000!
5
6 00000b 4C0A       RPT #10
7 00000d EF83       MOV *AR4+, coef(*CDP+) ; port of
  00000f 05                                ; MVDK *AR4+, name
```

To summarize, in the example above, the original C54x code:

```
                    RPT #10
                    MVDK *AR4+, name
```

was mapped to be:

```
                    AMAR *(#(name)),XCDP
                    RPT #10
                    MVDK *AR4+, coef(*CDP+)
```

Multiple-line mappings that do not require temporary registers are marked with a PORT comment.

A macro definition is simply echoed:

```
 8
 9            subm  .macro mem1, mem2, reg
10                 LD mem1, reg
11                 SUB mem2, reg
12                 .endm
```

A macro invocation is marked with a MACRO line. Within the macro expansion, you may see any of the cases described above.

```
13
14 ****** MACRO     subm name, *AR6, B
14 000010 A100%     LD name, B
14 000012 D7C1      SUB *AR6, B
   000014 11
```

Native C55x instructions appear without any special notation. For more information on using ported C54x code with native C55x code, see Section 7.3.

```
15
16 000015 2253      MOV T1, AC3 ; native C55x
```

6.3 Handling Reserved C55x Names

New C55x mnemonics and registers are reserved words. Your C54x code should not contain symbol names that are now used as C55x mnemonics or registers. For example, you should not use T3 as a symbol name.

Your C54x code also should not contain symbol names that are reserved words in the C55x algebraic syntax. For example, you should not have a label named return.

The C55x mnemonic assembler issues an error message when it encounters a symbol name conflict.

Migrating a C54x System to a C55x System

After you have ported your TMS320C54x code as described in Chapter 6, you must consider various system-level issues when moving your C54x code to the TMS320C55x. This chapter describes:

- How to handle differences related to interrupts
- How to use ported C54x functions with native C55x functions
- Non-portable C54x coding practices

Topic **Page**

7.1 Handling Interrupts

This section describes issues related to interrupts.

7.1.1 Differences in the Interrupt Vector Table

The C54x interrupt table is composed of 32 vectors. Each vector contains 4 words of executable code. The C55x vector table is also composed of 32 vectors. The vectors in both tables are the same length, but on the C55x, the length is counted as 8 bytes.

The order of the vectors in the interrupt vector table is documented in the data sheet for the specific device in your system. Since the order of the vectors is device-specific, any access to the IMR or IFR register needs to be updated accordingly. Likewise, if you use the TRAP instruction, its operand may need to be updated.

C54x and C55x handle the contents of their vectors in different ways. To handle these differences, you must modify the C54x vectors themselves.

In the C55x vector table, the first byte is ignored, and the next three bytes are interpreted as the address of the interrupt service routine (ISR). Use the .ivec assembler directive to initialize a C55x vector entry, as shown in the examples below. For more information on the .ivec directive, see Initialize Interrupt Table Entries.

Simple Branch to ISR

If the C54x vector contains:

B *isr*

Change the corresponding C55x vector to:

.ivec *isr*

Delayed Branch to ISR

If the C54x vector contains:

BD *isr*
inst_1 ; two instruction words of code
inst_2

The easiest solution is to write the vector as:

.ivec *isr*

and move the instructions inst1 and inst2 to the beginning of the ISR.

Vector Contains the Entire ISR

If the C54x vector contains the entire 4-word ISR, as in the examples shown below, you have to create the 4-word ISR as a stand-alone routine:

; example 1	; example 2	; example 3
inst1	*inst1*	**CALL**
inst2	**RETFD**	*routine1*
inst3	*inst2*	**RETE**
RETF	*inst3*	**nop**

You must then provide the address of that routine in the C55x vector table:

.ivec *new_isr*

7.1.2 Handling Interrupt Service Routines

An interrupt service routine needs to be changed only if when it is ported to C55x it includes C54x instructions that map to more than one C55x instruction, and one of the C55x instructions requires the use of a C55x register or bit as a temporary.

In this case, the new C55x register needs to be preserved by the routine.

See Section 7.3.2 for the list of C55x registers that can be used as temporaries in multiple-line instruction mappings.

To ensure that an interrupt will work, you can preserve the entire list of registers. Or, you can simply preserve the register(s) used:

- Assemble the ISR using cl55 with the --asm_listing option to generate a listing file.
- Check the listing to see if it includes a Temporary Registers Used comment at the top of the file, such as:

  ```
  16                  ; Temporary Registers Used: XCDP
  ```

  This comment provides a list of all temporary registers used in the porting of the file. For more information, see Section 6.2.
- If temporary registers are used, the appropriate register or bit must be pushed on the stack at the beginning of the ISR, and popped off the stack at the end.

7.1.3 Other Issues Related to Interrupts

You should be aware of the interrupt issues described below:

- When the assembler encounters RETE, RETED, FRETE, FRETED, RETF, or RETFD, a warning will be issued. With these instructions, the assembler is processing an interrupt service routine or the interrupt vector table itself and may not be able to port the instructions correctly.
- INTR has the same mnemonic syntax for both C54x and C55x. Consequently, the assembler cannot distinguish when an instruction is intended for a native C55x interrupt (which is acceptable) or for a C54x interrupt (for which the interrupt number may be wrong).
- If your code writes values to IPTR, a nine-bit field in the PMST indicating the location of the interrupt vector table, you will need to modify your code to reflect the changes in the C55x system.

7.2 Assembler Options for C54x Code

The cl55 assembler offers several options to provide additional support for the porting of C54x assembly code to C55x. With these options, the assembler can:

- Assume SST is disabled (-att option)
- Port for speed over size (-ath option)
- Encode for C54x-specific circular addressing (--purecirc option)
- Remove NOPs from delay slots (-atn option)

7.2.1 Assume SST Is Disabled (-att Option)

By default, the assembler assumes that the SST bit (saturate on store) is enabled. For example, the SST assumption causes the assembler to port the STH and STL instructions as follows:

C54x instruction	Default C55x encoding	Bytes
STH *src*, *Smem*	MOV HI(ACx << #0), *Smem*	3
STL *src*, *Smem*	MOV ACx << #0, *Smem*	3

The shift (<< #0) is used to achieve the same saturate-on-store behavior provided by C54x. Even if SST is disabled in your code, this encoding still works.

However, if the saturate behavior is not required, use the -mt assembler option to generate a more optimal encoding:

C54x instruction	Default C55x encoding with -att	Bytes
STH *src*, *Smem*	MOV HI(ACx), *Smem*	2
STL *src*, *Smem*	MOV ACx, *Smem*	2

The -att option affects the entire file. To toggle SST mode within a file, use the .sst_on and .sst_off assembler directives.

The .sst_on directive specifies that the SST status bit set to 1, the default assumption of the assembler. The .sst_off directive specifies that the SST status bit set to 0; this is equivalent to using the -mt assembler option. In the case of a conflict between the command line option and the directive, the directive takes precedence.

The scope of the .sst_on and .sst_off directives is static and not subject to the control flow of the assembly program. All of the assembly code between the .sst_off and the .sst_on directives is assembled with the assumption that SST is disabled. To indicate that the SST bit is disabled without using the command line option, place the .sst_off directive at the top of every source file.

7.2.2 Port for Speed Over Size (-ath Option)

By default, the assembler encodes C54x code with a goal of achieving small code size. For example, consider the encoding of the MVMM and STM instructions that write ARx registers. (In the STM instruction below, const is a constant in the range of -15 to 15.)

C54x instruction	Default C55x encoding	Bytes
MVMM AR*x*, AR*y*	MOV AR*x*, AR*y*	2
STM #const, AR*x*	MOV #*const*, AR*x*	2

You can use the -ath assembler option to generate a faster encoding:

C54x instruction	Default C55x encoding with -ath	Bytes
MVMM AR*x*, AR*y*	AMOV AR*x*, AR*y*	3
STM #const, AR*x*	AMOV #*const*, AR*y*	3

The MOV instruction writes ARy in the execute phase of the pipeline. AMOV writes ARy in the address phase, which is 4 cycles earlier. If the instruction following MVMM or STM de-references ARy (for example, *AR3+), MOV imposes a 4-cycle stall to wait for ARy to be written. AMOV does not impose a stall. The AMOV encoding provides a significant gain in speed at the cost of one byte of encoding space.

The -ath option affects the entire file. To toggle the port-for-speed mode within a file, use the .port_for_speed and .port_for_size assembler directives.

The .port_for_size directive models the default encoding of the assembler. The .port_for_speed directive models the effect of the -ath assembler option. In the case of a conflict between the command line option and the directive, the directive takes precedence.

Consider using .port_for_speed just before a critical loop. After the loop, use .port_for_size to return to the default encoding.

7.2.3 Optimized Encoding of C54x Circular Addressing (--purecirc Option)

If your ported C54x code uses C54x circular addressing without using the C55x linear/circular addressing bits, use the --purecirc option. This option allows the assembler to generate the most optimal encoding for the circular addressing code.

For the following example C54x code:

```
    RPTB   end-1
    NOP ; 1
    MAC    *AR5+, *AR3+0%, A
    NOP ; 2
end:
```

Building *without* --purecirc generates this code:

```
    RPTB   end-1
    NOP ; 1
    BSET   AR3LC
    MACM   T3 - *AR5+, *(AR3+AR0), AC0, AC0
    BCLR   AR3LC
    NOP ; 2
end:
```

Notice how the instructions for toggling the linear/circular bit for AR3 are still inside the loop. Building with --purecirc generates this code:

```
    BSET   AR3LC
    RPTB   P04_3
    NOP ; 1
    MACM   T3 - *AR5+, *(AR3+AR0), AC0, AC0
P04_3:
    NOP ; 2
    BCLR   AR3LC
end:
```

The instructions for toggling the linear/circular bit for AR3 are now outside of the loop.

Certain coding practices can hinder the optimization of circular addressing code, even when using the --purecirc option:

- Unused labels

  In the following code, the label middle is unused:

  ```
  start:
          RPTB   end-1
          LD     *AR4, A
  middle:                 ; unused label
          MAR    *AR4-0%
  end:
  ```

  If the unused label is removed from the loop, the assembler can move the circular bit operations for the MAR instruction out of the loop. Otherwise, the circular instructions remain in the loop, causing the loop to be 4 bytes larger and 4 cycles longer.

- Using a register for circular and non-circular purposes in the same loop.

  Consider the following code:

```
        RPTB    end-1
        ; reference to AR3 (circular)
        MAC     *AR5+, *AR3+0%, A

        ...

        ; reference to AR3 (non-circular)
        ST      A, *AR3+
   ||   SUB     *AR2, B

        ...
end:
```

  Because the second AR3 reference is non-circular, the circular bit operations of the MAC instruction cannot be moved outside of the loop. When possible, if one indirect reference of an ARx within a loop uses circular addressing, all indirect references of that register within that loop should also use circular addressing.

7.2.4 Removing NOPs in Delay Slots (-atn Option)

When the -atn option is specified, the assembler removes NOP instructions located in the delay slots of C54x delayed branch or call instructions.

For example, with the -atn option, the following C54x code:

```
CALLD  func
LD     *AR2, A
NOP
; call occurs here
```

will appear in the cl55 listing file as:

```
4 000000 A041   LD     *AR2, A
2
3 000002 6C00   CALLD  func
  000004 0000!
5 ****** DEL    NOP
6 ; call occurs here
```

The DEL in the opcode field signifies the deleted NOP.

7.3 Using Ported C54x Functions with Native C55x Functions

When rewriting a C54x application to be completely native C55x code, consider working on one function at a time, continually testing. If you encounter a problem, you can easily find it in the changes recently made. Throughout this process, you will be working with both ported C54x code and native C55x code. Keep the following in mind:

- Avoid mixing C54x and C55x instructions within the same function.
- Transitions between ported C54x instructions and native C55x instructions should occur only at function calls and returns.
- The C compiler provides the C54X_CALL pragma for C code calling assembly. However, see the example in Section 7.3.7 for a detailed description of using a veneer function when calling a ported C54x assembly function from C code. For more information on C54X_CALL, see the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.

7.3.1 Run-Time Environment for Ported C54x Code

A run-time environment is the set of presumptions and conventions that govern the use of machine resources such as registers, status register bit settings, and the stack. The run-time environment used by ported C54x code differs from the environment used by native C55x code. When you execute ported C54x code from reset, the appropriate run-time environment is already in place. However, when shifting from one kind of code to the other, it is important to be aware of the status bit and register settings that make up a particular environment.

The following CPU environment is expected upon entry to a ported C54x function:

- 32-bit stack mode
- The SP and SSP must be initialized to point into memory reserved for a stack. See Section 6.1.1.
- The status bits must be set as follows:

Status Bit	Set to...
C54CM	1
M40	0
ARMS	0
RDM	0
ST2[7:0] (circular addressing bits)	0

- The upper bits of addressing registers (DPH, CDPH, ARnH, SPH) must be set to 0.
- The BSA*xx* registers must be set to 0.

7.3.2 C55x Registers Used as Temporaries

The following C55x registers may be used as temporaries in multiple-line mappings generated by cl55:

- T0
- T1
- AC2
- AC3
- CDP
- CSR
- ST0_55 (TC1 bit only)
- ST2_55

Interrupt routines using these registers must save and restore them. For more information, see Section 7.1.2.

Native C55x code that calls ported C54x code must account for the possibility that ported code may overwrite these registers.

7.3.3 C54x to C55x Register Mapping

The following C54x registers map to C55x registers as shown below:

C54x	C55x
T	T3
A	AC0
B	AC1
AR*n*	AR*n*
IMR*n*	IER*n*
ASM (status bit in ST1)	T2

7.3.4 Caution on Using the T2 Register

Under the C54CM mode, which is required when running C54x code automatically ported by cl55, you cannot use the T2 register for any purpose other than to strictly model the ASM field of ST1 exactly as cl55 ported code does. Under C54CM, whenever the status register ST1_55 is written, the lower 5 bits (the ASM field) are automatically copied with sign extension to T2.

When an interrupt occurs, ST1_55 is automatically saved and restored. When the restore occurs, the automatic copy to T2 is restarted. Because of this automatic overwrite on the interrupt, you cannot use T2 as a general-purpose register even in sections of C54x code that do not use the ASM field.

7.3.5 Status Bit Field Mapping

The C55x status bit fields map to C54x status bit fields as shown below.

Table 7-1. ST0_55 Status Bit Field Mapping

Bit	C55x field	C54x field (in ST0)
15	ACOV2	none
14	ACOV3	none
13	TC1	none
12	TC2	TC
11	CARRY	C
10	ACOV0	OVA
9	ACOV1	OVB
8-0	DP	DP

Table 7-2. ST1_55 Status Bit Field Mapping

Bit	C55x field	C54x field (in ST0)
15	BRAF	BRAF
14	CPL	CPL
13	XF	XF
12	HM	HM
11	INTM	INTM
10	M40	none
9	SATD	OVM
8	SCMD	SXM
7	C16	C16
6	FRCT	FRCT
5	C54CM	none
4-0	ASM	ASM

Table 7-3. ST2_55 Status Bit Field Mapping

Bit	C55x field	C54x field (in ST0)
15	ARMS	none
14-13	Reserved	none
12	DBGM	none
11	EALLOW	none
10	RDM	none
9	Reserved	none
8	CDPLC	none
7-0	AR*n*LC	none

Table 7-4. ST3_55 Status Bit Field Mapping

Bit	C55x field	C54x field (in ST0)
15-8	Reserved	none
7	CBERR	none
6	MPNMC	MP/MC_
5	SATA	none
4	Reserved	none
3	Reserved	none
2	CLKOFF	CLKOFF
1	SMUL	SMUL
0	SST	SST

7.3.6 Switching Between Run-Time Environments

The run-time environment defined in Section 7.3.1 is not complete because it only defines registers and status bits that are new with C55x. Registers and status bits that are not new with C55x inherit their conventions from the original C54x code. (As shown in Section 7.3.3, some registers have new names.)

If the run-time environment for your native C55x code differs from the environment defined for ported C54x code, you must ensure that, when switching between environments, the proper adjustments are made for:

- Preserving status bit field values
- Preserving registers
- How arguments are passed
- How results are returned

Insert enviro54to55_pru280 image here.

7.3.7 Example of C Code Calling C54x Assembly

This example describes a technique for handling a call from compiled C code to a C54x assembly routine. In this example, an additional function is inserted between the native C55x code and the ported C54x code. This function, referred to as a veneer function, provides code to transition between the two run-time environments.

Compiler Pragmas

NOTE: The compiler provides two pragmas to do this work for you: C54X_CALL and C54X_FAR_CALL. If you use these pragmas, you do not need to write the veneer yourself. Both the C54x and C55x C compiler run-time environments are well-defined, which makes the techniques shown in this example more concrete and easier to apply to your own situation.

Example 7-1. C Prototype of Called Function

```
short firlat(short *x, short *k, short *r, short *dbuffer,
      unsigned short nx, unsigned short nk);
```

Example 7-2. Assembly Function _firlat_veneer

```
      .def   _firlat_veneer
      .ref   _firlat

_firlat_veneer:

; Saving Registers ------------------------
      PSH      AR5
          ; PSH   AR6      ; saved in ported C54x environment
      ; PSH   AR7      ; saved in ported C54x environment
      PSH      T2
      PSH      T3

; Passing Arguments ------------------------
      PSH      T1      ; push rightmost argument first
      PSH      T0      ; then the next rightmost
      PSH      AR3     ; and so on
      PSH      AR2
      PSH      AR1

      MOV      AR0, AC0   ; leftmost argument goes in AC0

; Change Status Bits -----------------------
      BSET   C54CM
      BCLR   ARMS
      BCLR   C16

; Call -------------------------------------
      CALL   _firlat

; Restore Status Bits ----------------------
      BCLR   C54CM
      BSET   ARMS
      BSET   SXMD

; Capture Result ---------------------------
      MOV      AC0, T0

; Clear Arguments From the Stack -----------
      AADD     #5, SP

; Restore Registers and Return -------------
      POP      T3
      POP      T2
      ; POP   AR7
      ; POP   AR6
      POP      AR5

      RET
```

The veneer function is described below. It is separated into several parts to allow for a description of each segment.

7.3.7.1 Saving Registers

```
PSH      AR5
; PSH   AR6 ; saved in ported C54x environment
; PSH   AR7 ; saved in ported C54x environment
PSH      T2
PSH      T3
```

If the C55x run-time environment expects that certain registers will not be modified by a function call, these registers must be saved. In the case of the C55x C compiler environment, registers XAR5-XAR7, T2, and T3 must be saved. Because C54x code cannot modify the upper bits of the XARn registers, only the lower bits need to be preserved. The instructions that push AR6 and AR7 are commented out because the run-time environment of the C54x ported code (as defined by the C54x C compiler) presumably saves these registers. A more conservative approach would be to save these registers anyway.

7.3.7.2 Passing Arguments

```
PSH      T1      ; push right-most argument first
PSH      T0      ; then the next argument
PSH      AR3     ; and so on
PSH      AR2
PSH      AR1
MOV      AR0, AC0  ; left-most argument goes in AC0
```

Arguments passed from native C55x code must be placed where the ported C54x code expects them. In this case, all arguments are passed in registers. According to the calling conventions of the C55x C compiler, the arguments to the firlat() function will be passed, and the result returned, in the registers shown below.

T0 AR0 AR1 AR2 AR3 T0 T1

```
short firlat(short *x, short *k, short *r, short *dbuffer, unsigned short nx, unsigned short nk);
```

For more information on the C compiler's calling conventions, see the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.

The ported C54x environment expects the first argument to be in A (AC0 on C55x) and the remaining arguments to be placed on the stack, in reverse order of appearance in the argument list. The right-most argument (T1) is pushed onto the stack first. The next argument (T0) is then pushed onto the stack. The argument placement continues until the left-most argument (AR0) is reached. This argument is copied to AC0.

7.3.7.3 Changing Status Bits

```
BSET   C54CM
BCLR   ARMS
BCLR   C16
```

It is necessary to change the status settings of the native C55x code to the settings required by ported C54x code. These settings are shown in Section 7.3.1. In this case, only the C54CM and ARMS bits need to be changed.

Because of the requirements for executing the original C54x code, it may be necessary to set the C16 bit to 0. This bit, ignored by C55x compiled code, is assumed to be 0 by the C54x compiler. Setting the bit to 0 is the conservative approach to account for this assumption.

7.3.7.4 Function Call

```
CALL   _firlat
```

Now that registers have been saved and status bits set, the call to ported C54x code can be made.

7.3.7.5 Restoring Status Bits

```
BCLR    C54CM
BSET    ARMS
BSET    SXMD
```

After the call, restore the status bits to the settings required by the native C55x environment. Ported C54x code makes no assumption about the SXMD bit (SXM on C54x) after a function call. However, C55x compiled code expects this bit to be set to 1.

7.3.7.6 Capturing Results

```
MOV     AC0, T0
```

The ported C54x environment returns the result in AC0, while the native C55x environment expects the result to be returned in T0. Consequently, the result must be copied from AC0 to T0.

7.3.7.7 Clearing Arguments From the Stack

```
AADD    #5, SP
```

At this point, you should decrease the stack by the number of words originally needed to push the function's passed arguments. In this case, the amount is 5 words. Because the stack grows from high addresses to low addresses, addition is used to change the stack pointer from a low address to a higher one.

7.3.7.8 Restoring Registers and Returning

```
POP     T3
POP     T2
; POP   AR7
; POP   AR6
POP     AR5

RET
```

Restore the registers saved at the beginning of the function, and return.

7.3.8 Example of C54x Assembly Calling C Code

Example 7-4 contains a C54x assembly routine calling a compiled C routine. Because the C routine is recompiled with the C55x C compiler, the assembly routine must handle the differences between the ported C54x run-time environment and the run-time environment used by the C55x compiler.

If you use a different run-time environment for your C55x code, your code changes will differ slightly from those in Example 7-3. However, you must still consider the issues addressed here.

Example 7-3. Prototype of Called C Function

```
int C_func(int *buffer, int length);
   ...
```

The assembly function performs some calculations not shown in this example and calls the C function. The returned result is copied to the C global variable named result. Further calculations, also not shown here, are then performed.

Example 7-4. Original C54x Assembly Function

```
; Declare some data -----------------
         .data
buffer:  .word 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
BUFLEN   .set 11
         .text

; Assembly routine starts -----------------

callsc:
; original C54x code ...

; Call C function (original C54x code) -----------------

         ST #BUFLEN, *SP(0)      ; pass 2nd arg on stack
         CALLD #_C_func
         LD #buffer, A           ; pass 1st arg in A

; Effects of calling C:
; May modify A, B, AR0, AR2-AR5, T, BRC
; Will not modify AR1, AR6, AR7
; May modify ASM, BRAF, C, OVA, OVB, SXM, TC
; Will not modify other status bits
; Presume CMPT = 0, CPL = 1

         STL A, *(_result)  ; Result is in accumulator A

; original C54x code ...

         RET
```

To use this assembly function on C55x, it is necessary to change the call to the C function.

Example 7-5. Modified Assembly Function

```
; declare data as shown previously

; Assembly routine starts -----------

callsc:
; ported C54x code ...

; Call C function (Change to C55x compiler environment)

         AMOV #buffer,AR0  ; pass 1st ptr arg in AR0
         MOV #BUFLEN,T0    ; pass 1st int arg in T0
; compiler code needs C54CM=0, ARMS=1
         BCLR C54CM ; clear C54x compatibility mode
         BSET ARMS ; set AR mode
         BSET SXM  ; set sign extension mode
         CALL _C_func ; no delayed call instruction

; Effects of calling C:
; May modify AC0-AC3, XAR0-XAR4, T0-T1
; May modify RPTC,CSR,BRCx,BRS1,RSAx,REAx
; Will not modify XAR5-XAR7,T2-T3,RETA
; May modify ACOV[013],CARRY,TC1,TC2,SATD,FRCT,ASM,
;  SATA,SMUL
; Will not modify other status bits

         MOV T0, *(_result) ; Result is in T0

; could use *abs16(_result) if all globals are in the
; same 64K word page of data

; Change back to ported C54x environment ----------
         BSET C54CM ; reset C54x compatibility mode
         BCLR ARMS ; disable AR mode

; ported C54x code ...

         RET
```

The arguments are passed according the calling conventions described in the *Run-Time Environment* chapter of the *TMS320C55x Optimizing C/C++ Compiler User's Guide*. The status bits modified are the only ones that differ between the C54x ported run-time environment and the native C55x environment (in this case, as defined by the C55x C compiler).

The comments about the effects of calling C (the registers and status bits that may or may not be modified) do not impact the code shown. But these effects can impact the code around such a call.

For example, consider the XAR1 register. In the C54x compiler environment, AR1 will not be modified by the call. In the C55x compiler environment, XAR1 may be modified. If code before the call to C_func loads a value into AR1, and code after the call reads AR1 for that value, then the code, as written, will not work on C55x. The best alternative is to use an XARn register that is saved by C routines, such as XAR5.

7.4 Output C55x Source

This section describes how to convert your C54x source code directly into C55x source code instead of object code. This conversion preserves your investment in C54x assembly code by using the same format, spacing, comments, and (very often) symbolic references of the original source.

7.4.1 Command-Line Options

Table 7-5 shows the cl55 command-line options.

Table 7-5. cl55 Command-Line Options

Option	Meaning
--mnem	Mnemonic output
--alg	Algebraic output
--incl	Write output for include files
--nomacx	Do not expand macros
--obj_directory=*directory*	Name the directory for the output files
--obj_extension=[.]*extension*	Name the extension of the output files

To get source output, you must use either --mnem or --alg. Otherwise, you produce the usual object files. Most of the examples in this section use cl55 --mnem even though cl55 --alg can be used.

If you do not specify an extension for the output files, the names of the output files are the same as those of the corresponding input files, but with a different extension. If the first letter of an input file extension is a or s, or there is no extension, the output file extension is *.s55*. Otherwise, the file is presumed to be an include file, and the output file extension is *.i55*.

For more information on --incl, see Section 7.4.2 and Section 7.4.3. For more information on --nomacx, see Section 7.4.7.

The following example shows that all of your assembly files are processed, placing the output in the directory c55x_asm, with the extension .asm instead of the default .s55. Any include files that are created are also in c55x_asm but named according to the default output file name method described above.

```
cl55 --quiet --mnem --incl --obj_directory=c55x_asm --obj_extension=.asm *.asm
```

The --obj_directory and --obj_extension Options

NOTE: The --obj_directory and --obj_extension options have the same meanings for .obj files when cl55 is used to compile and/or assemble files to object.

Since object files are not created, some compiler options that would normally affect the assembler do not apply. For instance, --asm_listing does not cause a listing file to be created.

Table 7-6 shows a list of the compiler options that affect the assembler. If you use an option listed as not having an effect, it is silently ignored.

Table 7-6. Compiler Options That Affect the Assembler

Option	Meaning	Effect
--absolute_listing	Enable absolute listing	No
-ar=*num*	Suppress remark *num*	Yes
--asm_define=*name*[=*def*]	Predefine *name*	Yes
--asm_listing	Produce asm listing file	No
--asm_undefine=*name*	Undefine *name*	Yes
-ata	Assert ARMS is initially set	No
-atc	Assert CPL is initially set	No
-ath	Port for speed over size	Yes
-atl	Assert C54x is initially set	No
-atn	Remove NOP in delay slots	Yes
-atp	Generate profiling .prf file	No
-att	Assert SST is always zero	Yes
-atv	All branches/calls are encoded as 24-bit offset	No
-atw	Suppress all warnings	Yes
--copy_file=*filename*	Copy the specified file for the assembly module	Yes
--cross_reference	Produce cross-reference file	No
--include_file=*filename*	Include the specified file for the assembly module	Yes
--output_all_syms	Keep local symbols	No
--syms_ignore_case	Make case insignificant	Yes

7.4.2 Processing .include/.copy Files

Only in this section, the term include file means a file included by either the .include or the .copy directive. An include file must be read in order to correctly process the file which includes it. This section addresses whether processing of an include file results in the creation of a corresponding output file.

By default, an output file is not written out for an include file and the .include statement itself remains unchanged. When the new .s55 file is re-assembled, it includes a file that has not been processed. Because the assembler can read C54x syntax, this does not affect correctness.

The --incl option changes this behavior. Under --incl, an output file is written out for each include file. The name of the new file is determined by the command-line option you use and the extension you specify as described in Section 7.4.1. Furthermore, the .include statement is modified to include the new output file.

For example, under --incl, the first statement below causes i1.i55 to be created, and the statement changes to the second statement below.

```
.include   i1.inc

.include   i1.i55
```

There is one special case on naming an include file. If cl55 is invoked with the --obj_directory option and the name of the include file does not contain any directory information, then the new include file is written out to the directory given by the --obj_directory option.

For example, the following statement places the new i1.i55 (and input.i55) file in the directory outdir.

```
cl55 --mnem --incl=input.asm --obj_directory=outdir
```

7.4.3 Problems with the --incl Option

Consider this contrived example of three files:

```
;  i1.inc-----------------------
     .word  x
; file1.asm-------------------
x    .set   0
     .include il.inc
; file2.asm-------------------
x    .set   1
     .include il.inc
```

Suppose you use --incl when building both file1.asm and file2.asm. Whether the new i1.i55 contains .word 0 or .word 1 depends on which file is built last. (It will certainly be wrong for one of them.)

The --incl option works only when every include file that is created is context free. That is, it contains no dependencies on the files which include it. In this case, i1.inc depends on the different values of x as defined in both file1.asm and file2.asm.

If i1.inc is included by several different files, using --incl causes i1.i55 to be written out each time cl55 --mnem is invoked on those files. Multiple developers working on different files in the same directory need to be aware that each time cl55 --mnem --incl is run, a new i1.i55 file is created.

Another problem with --incl relates to parallel makes. (If you don't know what a parallel make is, you can safely skip this paragraph.). Suppose you have a set of .asm files, which all include i1.inc. Furthermore, suppose you have a makefile that converts those files to C55x syntax with cl55 --mnem and then assembles the resulting .s55 files to object with just cl55. If you do this in parallel, you end up with simultaneous writes and reads to i1.i55. Since this does not work, you have to create the .i55 files with a serial make.

7.4.4 Handling .asg and .set

The .asg and .set lines are copied through unchanged. The use of symbols that are defined by .asg and .set is largely retained. Generally, if an entire operand can be copied unchanged from the old C54x instruction to the new C55x instruction, then that operand is copied through. But, if that operand is modified in any way, then the symbolic references may not show up.

7.4.5 Preserve Spacing with the .tab Directive

The assembler preserves the spacing of the original source line by copying it from the source file. However, when the width of a C55x mnemonic or operand field is wider than the original, some original spacing is omitted. To handle this step correctly, the assembler must know how many spaces are occupied by a tab. The default is 8 spaces. You can change this default with the directive **.tab** *size*, where *size* is how many spaces a tab occupies in your system.

7.4.6 Assembler-Generated Comments

Whenever source lines are deleted or added, the assembler uses a special prefix or suffix comment to mark these lines. These lines can then easily be found with the search features typically found in text editors, or scripting languages such as Perl and awk.

The general form of the comments is ";+XX" where XX is a two letter code used to specify the function the comment performs. These are described in the following sections.

7.4.6.1 Multiple-Line Rewrites

Multiple-line rewrites appear at the front of the commented out original source line. They are also tagged on the end of every line associated with the original source line.

The two types of multiple-line rewrites are:

- ML − Multiple-line rewrite
- RL - Multiple-line rewrite that uses a temporary register

7.4.6.2 Expanded Macro Invocations

Expanded macro invocations appear at the front of the macro invocation (always MI) or multiple line rewrite within the macro expansion. They are also tagged on the end of every line within the macro expansion.

The three types of expanded macro invocations are:

- MI − Single-line rewrite within an expanded macro
- MM - Multiple-line rewrite within an expanded macro
- RM - Multiple-line rewrite that uses a temporary register within an expanded macro

7.4.6.3 Prefix Comments

The following comments appear at the front of the commented out lines:

- NP - Deleted NOP
- IF - .if/.endif and related directives, as well as associated false blocks
- LP - .loop/.break/.endloop and enclosing lines
- FN - Naming the file
- MS - Miscellaneous

7.4.6.4 Suffix Comments

The following comment is tagged at the end of changed lines:

- SA − Converted .set to .asg (only in algebraic output)
- RK - Remark inserted with .mmsg directive. The RK comment is tagged at the end of added lines.

7.4.6.5 Code Example For Assembler-Generated Comments

Example 7-6 shows a code example of assembler-generated comments. Example 7-7 displays the C55x output for Example 7-6.

Example 7-6. Contrived C54x Assembly File

```
      .global name

      ADD     *AR2, A        ; same mnemonic
      LD      *AR3, B        ; different mnemonic

      RPT     #10
      MVDK    *AR4+, name    ; multi-line rewrite

subm  .macro  mem1, mem2, reg  ; macro definition copied through
      LD      mem1, reg
      SUB     mem2, reg
      .endm

      subm    name, *AR6, B    ; macro invocation expanded
      MOV 1, AC3             ; native LEAD3 instruction
```

Example 7-7. C55x Output For C54x Code Example in Example 7-6

```
;+MS translation of try1.asm
;+MS Temporary Registers Used: XCDP
      trans_count 1            ;+MS do NOT remove!
      global      name

      ADD      *AR2, AC0, AC0       ; same mnemonic
      MOV      *AR3, AC1            ; different mnemonic
;+RL  MVDK     *AR4+, name     ; multi-line rewrite
      AMAR     *(#name), XCDP  ; +RL port of MVDK *AR4+, name

      RPT      #10
      MOV      *AR4+, *CDP+    ; +RL port of MVDK *AR4+, name
subm  .macro   mem1, mem2, reg     ; macro definition copied through
      LD       mem1, reg
      SUB      mem2, reg
      .endm

;+MI  subm     name, *AR6, B       ; macro invocation expanded
      MOV      @#name, AC1    ;+MI
      SUB      *AR6, AC1, AC1 ;+MI

      MOV      T1, AC3        ; native LEAD3 instruction
```

7.4.7 Handling Macros

Macro definitions are always copied through unchanged.

By default, macro invocations are expanded. You can disable this expansion with the --nomacx option.

If you combine --alg --nomacx your output file has invocations of macros which use mnemonic syntax in a file that is otherwise algebraic syntax. Therefore, at the top of such a file you see the error message in Example 7-8.

Example 7-8. C55x Output Created from Combining --alg & --nomacx

```
.emsg "This file will not assemble because it combines algebraic syntax with
     invocations of macros in mnemonic syntax. Please see the comment at
     the top of <output file> for more information."
.end ;  stops assembler processing

Also, because this file cannot be assembled, this combination of features
cannot be tested. To attempt to assemble this file you must rewrite
the macros in C55x algebraic syntax, and remove this .emsg, .end, and
associated comment block.
```

If you assemble this file, you see the error message given in the .emsg. After the message is displayed, the assembler stops running. To continue, you must edit the file as instructed in the error message.

7.4.8 Handling the .if and .loop Directives

The problem with blocks of code controlled by the .if and .loop directives is that these blocks do not necessarily stay together through translation. Delayed branches or calls must move as part of the translation. If these delayed branches or calls occur just before such a block, they move into it. If they are near the end, they may leave the block.

The assembler will evaluate conditional expressions and comment out the .if, .else, .elseif, and .endif directives, as well as the code in the false block(s). The solution for .loop is to comment out every thing from the .loop to the .endloop, including any .break directives, and follow that with as many iterations of the .loop block as required.

Loop Count Affects Translated Source Size

NOTE: If the loop count is a large value, then there will be a large increase in the size of the translated source versus the original source.

7.4.9 Integration Within Code Composer Studio

Converting source from C54x to C55x is not a process integrated within Code Composer Studio (CCStudio). None of the command-line options described in this section are available from within CCStudio. Use the command line interface to cl55 to convert your C54x source to C55x, then add those new C55x source files to your C55x CCStudio project.

7.5 Non-Portable C54x Coding Practices

Some C54x coding practices cannot be ported to the C55x. The assembler warns you of certain detectable issues, but it cannot detect every issue. The following coding practices are not portable:

- Any use of a constant as a memory address. For example:

```
B  42
ADD @42,A
SUB @symbol+10,b
```

- Memory initialized with constants that are later interpreted as code addresses. For example:

```
table: .word 10, 20, 30
       ...
       LD    @table,A
       CALA
```

- Using data as instructions. For example:

```
function:
      .word 0xabcd  ; opcode for ???
      .word 0xdef0  ; opcode for ???
...
      CALL function
```

- Out-of-order execution, also known as pipeline tricking. The assembler detects one instance of out-of-order execution: when an instruction modifies the condition in the two instruction-words before the C54x XC instruction. In this instance, the assembler issues a remark. Other cases of out-of order execution are not detected by the assembler.
- Code that creates or modifies code.
- Repeat blocks spanning more than one file.
- Branching/calling unlabeled locations. Or, modifying the return address to return to unlabeled location. This includes instructions such as:

```
B   $+10
```

- Using READA and WRITA instructions to access instructions and not data. For more information, see Section 7.6.
- Using READA/WRITA with an accumulator whose upper bits are not zero.

  The READA/WRITA instruction on C54x devices (other than C548 or later) uses the lower 16 bits of the accumulator and ignores the upper 16 bits. C548 and later devices, however, use the lower 23 bits.

The assembler assumes C548 or later devices as it cannot determine the device for which the code is targeted. Consequently, code for C548 and later devices maps with no problems. Code for devices other than these does not run.

- Label differences are not allowed in conditional assembly expressions.

7.6 Additional C54x Issues

There are some additional system issues.

If your C54x code does any of the following, you may need to modify this code to use native C55x instructions:

- Uses a *SP(offset) operand in the MMR slot of MMR instructions like LDM
- Copies blocks of code, usually from off-chip memory to on-chip memory
- Uses memory-mapped access to peripherals
- Uses repeat blocks larger than 32K after mapping to C55x
- Uses the branch conditions BIO/NBIO

7.6.1 C54x to C55x Incompatibilities

You should also be aware of the following issues:

- The C5x-compatibility features of the C54x are not supported on C55x.
- RPT instructions, non-interruptible on C54x, can be interrupted on C55x.
- When an operation overflows into the guard bits, and then a left-shift clears the guard bits, the C54x has the value of zero while the C55x has a saturated value.
- The C54x and C55x mnemonic assembly languages differ significantly in the representation of instruction parallelism.

  The C55x implements two types of parallelism: implied parallelism within a single instruction (using the :: operator), and user-defined parallelism between two instructions (using the || operator). The C54x implements only one type of parallelism, which is analogous to implied parallelism on the C55x. However, C54x parallelism uses parallel bars (||) as its operator. C55x parallelism is documented in the *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*.
- When using indirect access with memory-mapped access instructions, such as the following, the C54x masks the upper 9 bits of the ARn register.

```
   STM  #0x1234, *AR2+
```

  This masking effectively occurs both before and after the post-increment to AR2. For example:

```
; AR2 = 0x127f
   STM  #0x1234, *AR2+ ; access location 0x7f
; AR2 = (0x7f + 1) & ~7f ==> 0
```

  However, the C55x assembler maps this as follows to account for the possibility of a memory-mapped address for AR2.

```
   AND  #0x7f, AR2
   MOV  #0x1234, *AR2+ ; note no masking afterward
```

7.6.2 Handling Program Memory Accesses

The cl55 assembler supports C54x program memory access instructions (FIRS, MACD, MACP, MVDP, MVPD, READA, WRITA) for accessing data, but not for accessing code. When the assembler encounters one of these instructions, it will issue a remark (R5017). On C54x, a code address is in words, while on C55x, it is in bytes. To account for this difference when handling program memory access instructions, the assembler does the following actions:

- Generates a C55x instruction sequence with the assumption that the C54x program memory access operand refers to a data (word) address, not a code (byte) address.
- Places any data declaration found in a code section into its own data section. This will most likely require changes to your linker command file.

For example, the C54x input in Example 7-9 is ported by C55x to the code in Example 7-10.

Example 7-9. C54x Input

```
      .global ext
      MVDP    *AR2, ext
table:
      .word 10
```

Example 7-10. C55x Result

```
      .global ext
      AMOV    #ext, XCDP
      MOV     *AR2, *CDP
      .sect   ".data:.text"
table:
      .word 10
```

In Example 7-10, the instructions generated for MVDP assume that ext is a data (word) address. If the memory address used in your code actually is a code address, the C55x instructions will not work. In this case, you should rewrite the function to use native C55x instructions. For more information on using native C55x instructions along with ported C54x code, see Section 7.3.

The .word directive in this example is placed into a new section called .data:.text. In general, groupings of data within a code section are placed into subsections with the name .data:root_section, where root_section is the name of the original code section used on C54x. Your linker command file should be modified to account for these changes. A subsection can be allocated separately or grouped with other sections using the same base name. For example, to group all data sections and subsections:

```
.data > RAM ; allocates all .data sections / subsections
```

For more information on subsections, see Section 2.2.4.

7.7 Assembler Messages

When assembling C54x code, cl55 may generate any of the following remarks. To suppress a particular remark or all remarks, use the -ar assembler option or the .noremark directive. For more information, see Control Remarks.

(R5001) Possible dependence in delay slot of RPTBD--be sure delay instructions do not modify repeat control registers.

Description This message occurs when the instructions in the delay slots of a C54x RPTBD instruction perform indirect memory references.

Action If these instructions modify the REA or RSA repeat address control registers, the C55x instructions used to implement RPTBD will not work. If the instructions do not modify REA or RSA, you can either ignore this message or rewrite your code to use RTPB.

(R5002) Ignoring RSBX CMPT instruction

Description This C54x instruction disables the C5x compatibility mode of the C54x. Because C55x does not support C5x compatibility mode, this instruction is ignored.

Action Remove this instruction from your code, or simply ignore this message.

(R5003) C54x does not modify ARn, but C55x does

Description This message occurs when both memory operands of an ADD or SUB instruction use the same ARn register but only the second operand modifies the register. For example

```
SUB *AR3, *AR3+, A
```

Action On C54x, such an instruction will not modify AR3 by adding one to it. On C55x, the same instruction will add one to AR3.This difference in behavior may or may not affect your code. To prevent this message from being issued, move the ARn modification to the first operand:

```
SUB *AR3+, *AR3, A
```

(R5004) Port of RETF correct only for non-interrupt routine.

Description This message occurs when the assembler encounters RETF and RETFD, the C54x fast interrupt return instructions. Because it is possible to correctly use these instructions in non-interrupt routines, the RETF instruction is mapped to the C55x RET instruction.

Action If this instance of RETF or RETFD is actually used to return from an interrupt, you need to consider the issues described in R5005, and then rewrite this instruction using the C55x RETI instruction.

(R005) Port of [F]RETE is probably not correct. Consider rewriting to use RETI instead.

Description This message occurs when the assembler encounters the C54x RETE, RETED, FRETE, and FRETED instructions. These instructions are mapped to the C55x RETI instruction.

Action The effects of RETI differ from the effects of the RETE instructions. For example, RETI automatically restores ST1_55, ST2_55, and part of ST0_55. RETE does not. You may need to adjust your code accordingly. Furthermore, you need to determine if your C54x interrupt service routine contains any multiple-line mappings using C55x temporary registers. If so, you need to preserve the registers. For more information, see Section 7.1.2.

(R006) This instruction loads the memory address itself, and not the contents at that memory address

Description This message occurs when the first operand of an AMOV instruction is a symbol without an operand prefix. For example:

```
AMOV symbol, XAR3 ; not written as #symbol
```

Action This instruction may seem to load the contents at the memory address represented by symbol. However, the address of the symbol itself is loaded. Use the # prefix to correct this issue:

```
AMOV #symbol, XAR3
```

(R007) C54x and C55x port numbers are different

Description This message occurs when the assembler encounters C54x PORTR and PORTW instructions. A C55x instruction sequence will be encoded to perform the same function, but the port number used will most likely be incorrect for C55x.

Action Consider rewriting the code to use a similar C55x instruction that loads/stores the contents of a port address into a register:

```
MOV port(#100), AC0 ; for PORTR MOV AC1, port(#200) ; for PORTW
```

(R008) C54x directive ignored

Description Some C54x assembler directives are not needed on the C55x. This message occurs when you use such a directive (.version, .c_mode, .far_mode).

Action Remove this directive from your code, or simply ignore this message.

(R009) Modifying C54x IPTR in PMST will not update C55x IVPD/ IVPH. Replace with native C55x mnemonic (e.g., MOV #K, mmap(IVPD)).

Description This message occurs when the assembler encounters a write to the PMST register. On C54x, bits 15 through 7 of PMST contain the upper 9 bits of the address of the interrupt vector table. C55x uses the IVPD/IVPH registers for this role. The IVPD/IVPH registers are described in the *TMS320C55x DSP CPU Reference Guide*.

Action Replace the C54x instruction with a native C55x instruction.

(R010) C54x and C55x interrupt enable/flag registers and bit mapping are different. Replace with native C55x mnemonic.

Description This message occurs when the assembler encounters a write to the IFR or IMR registers. The bit mappings of the C55x IFR and IER (IMR on C54x) registers differ from the C54x mappings. These registers are described in the *TMS320C55x DSP CPU Reference Guide*.

Action Replace the C54x instruction with a native C55x instruction.

(R011) C55x requires setting up the system stack pointer (SSP) along with the usual C54x SP setup.

Description This message occurs when the assembler encounters a write to the SP register. C55x has a primary system stack managed by the SP as well as a secondary system stack managed by SSP. This remark is a reminder that whenever SP is initialized, SSP must be initialized also.

Action Initialize the SSP register.

(R012) This instruction requires the use of C55x 32-bit stack mode.

Description This message occurs when the assembler encounters the FCALL[D] or FCALA[D] instructions. These instructions only work in 32-bit stack mode. The stack configurations are described in the *TMS320C55x DSP CPU Reference Guide*. Since 32-bit stack mode is the default mode upon device reset, you must explicitly set up your reset vector to use a different stack mode. For more information, see Initialize Interrupt Table Entries.

Action Set the stack configuration accordingly.

(R013) C55x peripheral registers are in I/O space. Use C55x port() qualifier.

Description This message occurs when the assembler encounters the use of a C54x peripheral register name. These registers are not memory-mapped on C55x. Instead, they are located in I/O space. To access C55x I/O space, you must use the port() operand qualifier. For more information, see the *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*.

Action Use the port() qualifier accordingly.

(R014) On C54x, the condition set in the two instruction words before an XC does not affect that XC. The opposite is true on C55x.

Description This message occurs when the assembler encounters an instruction that modifies the condition in the two instruction words before the C54x XC instruction. On C54x, this code depends on out-of-order execution in the pipeline. However, this out-of-order execution will not occur on the C55x, so the results will not be the same. Out-of-order execution is considered a non-portable C54x coding practice, as described in Section 7.5. While there are many possible cases of out-of-order execution, this is the only one detected by the assembler.

Action Modify your code to account for the difference on C55x.

(R015) Using hard-coded address for branch/call destination is not portable from C54x.

Description This message occurs when the assembler encounters a C54x instruction that includes a branch or call to a non-symbolic, hard-coded address. Because code addresses are words on C54x and bytes on C55x, the assembler cannot know if the address accounts for the byte/word difference.

Action Modify your code to account for the difference on C55x.

(R016) Using expression for branch/call destination is not portable from C54x.

Description This message occurs when the assembler encounters a C54x branch or call instruction with an expression containing an arithmetic operator (such as sym+1). Because code addresses are words on C54x and bytes on C55x, the assembler cannot know if your code accounts for the byte/ word difference.

Action Modify your code to account for the difference on C55x.

(R017) Program memory access is supported when accessing data, but not when accessing code. In addition, changes to your linker command file are typically required.

Description This message occurs when the assembler encounters a C54x program memory access instruction (FIRS, MACD, MACP, MVDP, MVPD, READA, WRITA). For more information, see Section 7.6.2.

Action Modify your code and/or linker command file to account for the C55x differences.

- Built-in parallelism within a single instruction.

  Some instructions perform two different operations in parallel. Double colons (::) are used to separate the two operations. This type of parallelism is also called implied parallelism. These instructions are provided directly by the device and are documented in the *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*. You cannot form your own implied parallel instructions.

- User-defined parallelism between two independent instructions.

  Two instructions may be paralleled by you, as allowed by the parallelism rules described in the *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*. Parallel bars (||) are used to separate two instructions to be executed in parallel.

The C54x implements only one type of parallelism. It is analogous to implied parallelism on the C55x. However, C54x parallelism uses parallel bars (||) as its operator.

Table 7-7 summarizes the parallelism operators on the C54x and C55x.

Table 7-7. Parallelism Operators

Kind of parallelism	C54x Operator	C55x Operator
Implied	||	::
User-defined	N/A	||

Archiver Description

The TMS320C55x archiver lets you combine several individual files into a single archive file. For example, you can collect several macros into a macro library. The assembler searches the library and uses the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members.

Topic **Page**

8.1 Archiver Overview

You can build libraries from any type of files. Both the assembler and the linker accept archive libraries as input; the assembler can use libraries that contain individual source files, and the linker can use libraries that contain individual object files.

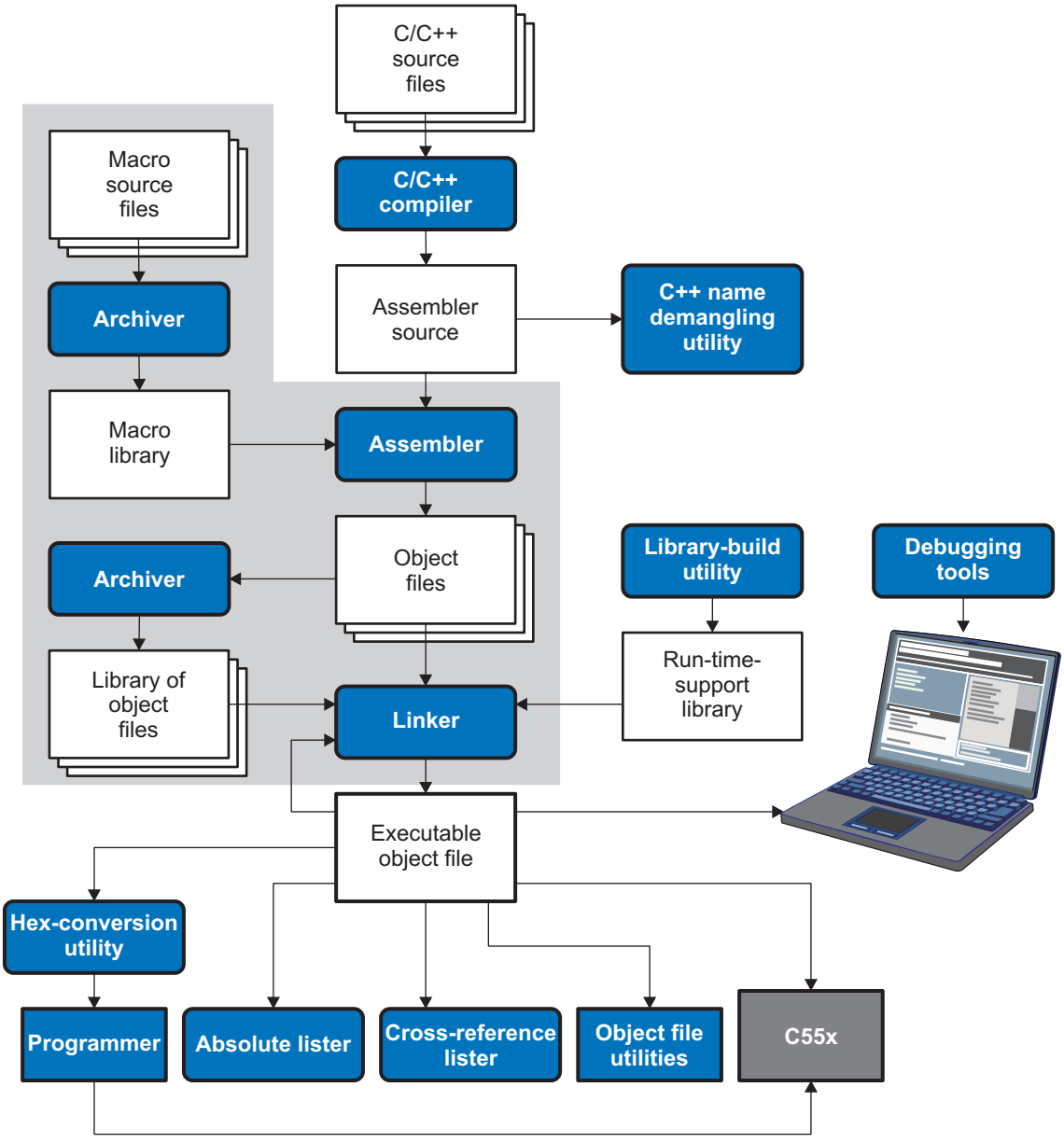
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The linker searches the library and includes members that resolve external references.

You can also use the archiver to build macro libraries. You can create several source files, each of which contains a single macro, and use the archiver to collect these macros into a single, functional group. You can use the .mlib directive during assembly to specify that macro library to be searched for the macros that you call. Chapter 5 discusses macros and macro libraries in detail, while this chapter explains how to use the archiver to build libraries.

8.2 The Archiver's Role in the Software Development Flow

Figure 8-1 shows the archiver's role in the software development process. The shaded portion highlights the most common archiver development path. Both the assembler and the linker accept libraries as input.

Figure 8-1. The Archiver in the TMS320C55x Software Development Flow



8.3 Invoking the Archiver

To invoke the archiver, enter:

ar55 [-]*command* [*options*] *libname* [*filename*$_1$ ... *filename*$_n$]

ar55 is the command that invokes the archiver.

[-]*command* tells the archiver how to manipulate the existing library members and any specified. A command can be preceded by an optional hyphen. You must use one of the following commands when you invoke the archiver, but you can use only one command per invocation. The archiver commands are as follows:

- **@** uses the contents of the specified file instead of command line entries. You can use this command to avoid limitations on command line length imposed by the host operating system. Use a ; at the beginning of a line in the command file to include comments. (See Example 8-1 for an example using an archiver command file.)
- **a** adds the specified files to the library. This command does not replace an existing member that has the same name as an added file; it simply *appends* new members to the end of the archive.
- **d** deletes the specified members from the library.
- **r** replaces the specified members in the library. If you do not specify filenames, the archiver replaces the library members with files of the same name in the current directory. If the specified file is not found in the library, the archiver adds it instead of replacing it.
- **t** prints a table of contents of the library. If you specify filenames, only those files are listed. If you do not specify any filenames, the archiver lists all the members in the specified library.
- **x** extracts the specified files. If you do not specify member names, the archiver extracts all library members. When the archiver extracts a member, it simply copies the member into the current directory; it *does not* remove it from the library.

options In addition to one of the *commands,* you can specify options. To use options, combine them with a command; for example, to use the a command and the s option, enter -as or as. The hyphen is optional for archiver options only. These are the archiver options:

- **-q** (quiet) suppresses the banner and status messages.
- **-s** prints a list of the global symbols that are defined in the library. (This option is valid only with the a, r, and d commands.)
- **-u** replaces library members only if the replacement has a more recent modification date. You must use the r command with the -u option to specify which members to replace.
- **-v** (verbose) provides a file-by-file description of the creation of a new library from an old library and its members.

libname names the archive library to be built or modified. If you do not specify an extension for *libname,* the archiver uses the default extension *.lib*.

filenames names individual files to be manipulated. These files can be existing library members or new files to be added to the library. When you enter a filename, you must enter a complete filename including extension, if applicable.

Naming Library Members

NOTE: It is possible (but not desirable) for a library to contain several members with the same name. If you attempt to delete, replace, or extract a member whose name is the same as another library member, the archiver deletes, replaces, or extracts the first library member with that name.

8.4 Archiver Examples

The following are examples of typical archiver operations:

- If you want to create a library called function.lib that contains the files sine.obj, cos.obj, and flt.obj, enter:

```
ar55 -a function sine.obj cos.obj flt.obj
```

  The archiver responds as follows:

```
==> new archive 'function.lib' ==> building new archive 'function.lib'
```

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

```
ar55 -t function
```

  The archiver responds as follows:

```
          FILE NAME    SIZE   DATE
   ---------------   -----   -----------------------
           sine.obj     300   Wed Jun 15 10:00:24 2011
            cos.obj     300   Wed Jun 15 10:00:30 2011
            flt.obj     300   Wed Jun 15 09:59:56 2011
```

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

```
ar55 -as function atan.obj
```

  The archiver responds as follows:

```
==> symbol defined: '_sin'
==> symbol defined: '_cos'
==> symbol defined: '_tan'
==> symbol defined: '_atan
==> building archive 'function.lib'
```

  Because this example does not specify an extension for the libname, the archiver adds the files to the library called function.lib. If function.lib does not exist, the archiver creates it. (The -s option tells the archiver to list the global symbols that are defined in the library.)

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

```
ar55 -x macros push.asm
```

  The archiver makes a copy of push.asm and places it in the current directory; it does not remove push.asm from the library. Now you can edit the extracted file. To replace the copy of push.asm in the library with the edited copy, enter:

```
ar55 -r macros push.asm
```

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

```
ar55 -@modules.cmd
```

  The archiver responds as follows:

```
==>  building archive 'modules.lib'
```

  Example 8-1 is the modules.cmd command file. The r command specifies that the filenames given in the command file replace files of the same name in the modules.lib library. The -u option specifies that these files are replaced only when the current file has a more recent revision date than the file that is in the library.

Example 8-1. Archiver Command File

```
; Command file to replace members of the
;     modules library with updated files
; Use r command and u option:
ru
; Specify library name:
modules.lib
; List filenames to be replaced if updated:
align.asm
bss.asm
data.asm
text.asm
sect.asm
clink.asm
copy.asm
double.asm
drnolist.asm
emsg.asm
end.asm
```

8.5 Library Information Archiver Description

Section 8.1 explains how to use the archiver to create libraries of object files for use in the linker of one or more applications. You can have multiple versions of the same object file libraries, each built with different sets of build options. For example, you might have different versions of your object file library for big and little endian, for different architecture revisions, or for different ABIs depending on the typical build environments of client applications. Unfortunately, if there are several different versions of your library it can become cumbersome to keep track of which version of the library needs to be linked in for a particular application.

When several versions of a single library are available, the library information archiver can be used to create an index library of all of the object file library versions. This index library is used in the linker in place of a particular version of your object file library. The linker looks at the build options of the application being linked, and uses the specified index library to determine which version of your object file library to include in the linker. If one or more compatible libraries were found in the index library, the most suitable compatible library is linked in for your application.

8.5.1 Invoking the Library Information Archiver

To invoke the library information archiver, enter:

libinfo55 [*options*] **-o**=*libname libname*$_1$ [*libname*$_2$ ... *libname*$_n$]

libinfo55 is the command that invokes the library information archiver.

options changes the default behavior of the library information archiver. These options are:

- **-o** *libname* specifies the name of the index library to create or update. This option is required.
- **-u** updates any existing information in the index library specified with the -o option instead of creating a new index.

libnames names individual object file libraries to be manipulated. When you enter a libname, you must enter a complete filename including extension, if applicable.

8.5.2 Library Information Archiver Example

Consider these object file libraries that all have the same members, but are built with different build options:

Object File Library Name	Build Options

Using the library information archiver, you can create an index library called mylib.lib from the above libraries:

You can now specify mylib.lib as a library for the linker of an application. The linker uses the index library to choose the appropriate version of the library to use. If the --issue_remarks option is specified before the --run_linker option, the linker reports which library was chosen.

8.5.3 Listing the Contents of an Index Library

The archiver's -t option can be used on an index library to list the archives indexed by an index library:

The indexed object file libraries have an additional .libinfo extension in the archiver listing. The __TI_$$LIBINFO member is a special member that designates *mylib.lib* as an index library, rather than a regular library.

If the archiver's -d command is used on an index library to delete a .libinfo member, the linker will no longer choose the corresponding library when the index library is specified.

Using any other archiver option with an index library, or using -d to remove the __TI_$$LIBINFO member, results in undefined behavior, and is not supported.

8.5.4 Requirements

You must follow these requirements to use library index files:

- At least one of the application's object files must appear on the linker command line before the index library.
- Each object file library specified as input to the library information archiver must only contain object file members that are built with the same build options.
- The linker expects the index library and all of the libraries it indexes to be in a single directory.

Linker Description

The TMS320C55x linker creates executable modules by combining object modules. This chapter describes the linker options, directives, and statements used to create executable modules. Object libraries, command files, and other key concepts are discussed as well.

The concept of sections is basic to linker operation; Chapter 2 discusses the object module sections in detail.

Topic **Page**

9.1 Linker Overview

The TMS320C55x linker allows you to configure system memory by allocating output sections efficiently into the memory map. As the linker combines object files, it performs the following tasks:

- Allocates sections into the target system's configured memory
- Relocates symbols and sections to assign them to final addresses
- Resolves undefined external references between input files

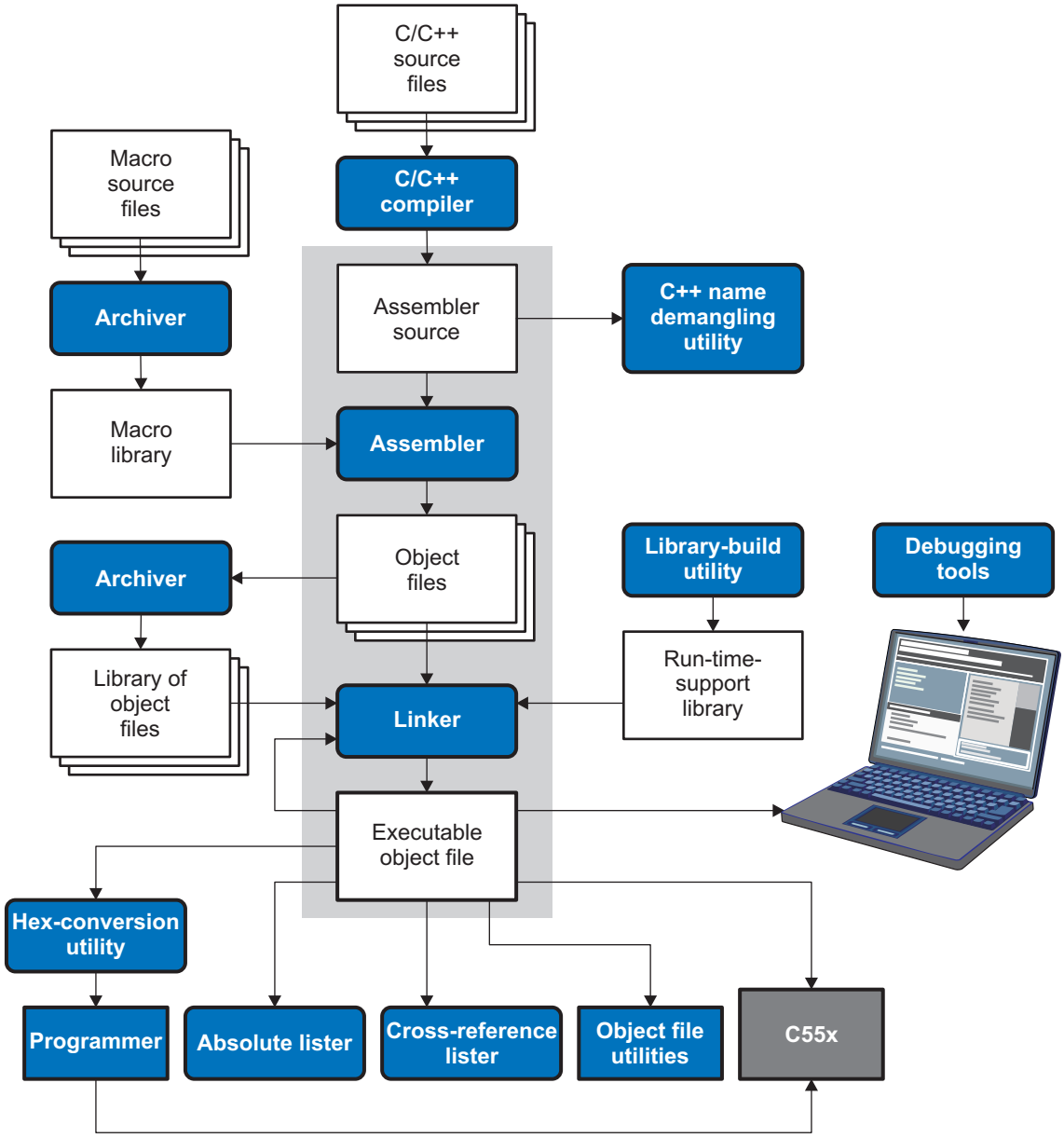
The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow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

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

9.2 The Linker's Role in the Software Development Flow

Figure 9-1 illustrates the linker's role in the software development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable object module that can be downloaded to one of several development tools or executed by a TMS320C55x device.

Figure 9-1. The Linker in the TMS320C55x Software Development Flow



9.3 Invoking the Linker

The general syntax for invoking the linker is:

cl55 --run_linker [*options*] *filename*$_1$ .... *filename*$_n$

cl55 --run_linker	is the command that invokes the linker. The --run_linker option's short form is -z.
options	can appear anywhere on the command line or in a link command file. (Options are discussed in Section 9.4.)
filename $_1$, *filename* $_n$	can be object files, link command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the --output_file option to name the output file.

There are two methods for invoking the linker:

- Specify options and filenames on the command line. This example links two files, file1.obj and file2.obj, and creates an output module named link.out.

  ```
  cl55 --run_linker file1.obj file2.obj --output_file=link.out
  ```

- Put filenames and options in a link command file. Filenames that are specified inside a link command file must begin with a letter. For example, assume the file linker.cmd contains the following lines:

  ```
  --output_file=link.out file1.obj file2.obj
  ```

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

  ```
  cl55 --run_linker 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:

  ```
  cl55 --run_linker --map_file=link.map linker.cmd file3.obj
  ```

  The linker reads and processes a command file as soon as it encounters the filename on the command line, so it links the files in this order: file1.obj, file2.obj, and file3.obj. This example creates an output file called link.out and a map file called link.map.

For information on invoking the linker for C/C++ files, see Section 9.11.

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 (-). Options can be separated from arguments (if they have them) by an optional space. Table 9-1 summarizes the linker options.

Table 9-1. Basic Options Summary

Option	Alias	Description	Section
--output_file	-o	Names the executable output module. The default filename is a.out.	Section 9.4.19
--map_file	-m	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename*	Section 9.4.14
--stack_size	-stack	Sets primary C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 1K bytes	Section 9.4.23
--heap_size	-heap	Sets heap size (for the dynamic memory allocation in C) to *size* bytes and defines a global symbol that specifies the heap size. Default = 2K bytes	Section 9.4.10

Table 9-2. File Search Path Options Summary

Option	Alias	Description	Section
--library	-l	Names an archive library or link command *filename* as linker input	Section 9.4.12
--search_path	-i	Alters library-search algorithms to look in a directory named with *pathname* before looking in the default location. This option must appear before the --library option.	Section 9.4.12.1
--priority	-priority	Satisfies unresolved references by the first library that contains a definition for that symbol	Section 9.4.12.3
--reread_libs	-x	Forces rereading of libraries, which resolves back references	Section 9.4.12.3
--disable_auto_rts		Disables the automatic selection of a run-time-support library	Section 9.4.5

Table 9-3. Command File Preprocessing Options Summary

Option	Alias	Description	Section
--define		Predefines *name* as a preprocessor macro.	Section 9.4.7
--undefine		Removes the preprocessor macro *name*.	Section 9.4.7
--disable_pp		Disables preprocessing for command files	Section 9.4.7

Table 9-4. Diagnostic Options Summary

Option	Alias	Description	Section
--diag_error		Categorizes the diagnostic identified by *num* as an error	Section 9.4.4
--diag_remark		Categorizes the diagnostic identified by *num* as a remark	Section 9.4.4
--diag_suppress		Suppresses the diagnostic identified by *num*	Section 9.4.4
--diag_warning		Categorizes the diagnostic identified by *num* as a warning	Section 9.4.4
--display_error_number		Displays a diagnostic's identifiers along with its text	Section 9.4.4
--emit_warnings_as_errors	-pdew	Treats warnings as errors	Section 9.4.4
--issue_remarks		Issues remarks (nonserious warnings)	Section 9.4.4
--no_demangle		Disables demangling of symbol names in diagnostics	Section 9.4.16
--no_warnings		Suppresses warning diagnostics (errors are still issued)	Section 9.4.4
--set_error_limit		Sets the error limit to *num*. The linker abandons linking after this number of errors. (The default is 100.)	Section 9.4.4
--verbose_diagnostics		Provides verbose diagnostics that display the original source with line-wrap	Section 9.4.4
--warn_sections	-w	Displays a message when an undefined output section is created	Section 9.4.28

Table 9-5. Linker Output Options Summary

Option	Alias	Description	Section
--absolute_exe	-a	Produces an absolute, executable module. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe were specified.	Section 9.4.2.1
--mapfile_contents		Controls the information that appears in the map file.	Section 9.4.15
--relocatable	-r	Produces a nonexecutable, relocatable output module	Section 9.4.2.2
--rom	-r	Create a ROM object	
--run_abs	-abs	Produces an absolute listing file	Section 9.4.21
--xml_link_info		Generates a well-formed XML *file* containing detailed information about the result of a link	Section 9.4.29

Table 9-6. Symbol Management Options Summary

Option	Alias	Description	Section
--entry_point	-e	Defines a global symbol that specifies the primary entry point for the output module	Section 9.4.8
--globalize		Changes the symbol linkage to global for symbols that match *pattern*	Section 9.4.13
--hide		Hides global symbols that match *pattern*	Section 9.4.11
--localize		Changes the symbol linkage to local for symbols that match *pattern*	Section 9.4.13
--make_global	-g	Makes *symbol* global (overrides -h)	Section 9.4.13.2
--make_static	-h	Makes all global symbols static	Section 9.4.13.1
--no_sym_merge	-b	Disables merge of symbolic debugging information in COFF object files	Section 9.4.17
--no_symtable	-s	Strips symbol table information and line number entries from the output module	Section 9.4.18
--scan_libraries	-scanlibs	Scans all libraries for duplicate symbol definitions	Section 9.4.22
--symbol_map		Maps symbol references to a symbol definition of a different name	Section 9.4.25
--undef_sym	-u	Places an unresolved external *symbol* into the output module's symbol table	Section 9.4.27
--unhide		Reveals (un-hides) global symbols that match *pattern*	Section 9.4.11

Table 9-7. Run-Time Environment Options Summary

Option	Alias	Description	Section
--arg_size	--args	Allocates memory to be used by the loader to pass arguments	Section 9.4.3
--fill_value	-f	Sets default fill values for holes within output sections; *fill_value* is a 32-bit constant	Section 9.4.9
--ram_model	-cr	Initializes variables at load time	Section 9.4.20
--rom_model	-c	Autoinitializes variables at run time	Section 9.4.20
--sysstack	-sysstack	Sets the secondary system stack size to *size* bytes and defines a global symbol that specifies the secondary stack size. Default = 1K bytes	Section 9.4.26

Table 9-8. Miscellaneous Options Summary

Option	Alias	Description	Section
--disable_clink	-j	Disables conditional linking of COFF object modules	Section 9.4.6
--linker_help	-help	Displays information about syntax and available options	–
--preferred_order		Prioritizes placement of functions	
--strict_compatibility		Performs more conservative and rigorous compatibility checking of input object files	Section 9.4.24

9.4.1 Wild Cards in File, Section, and Symbol Patterns

The linker allows file, section, and symbol names to be specified using the asterisk (*) and question mark (?) wild cards. Using * matches any number of characters and using ? matches a single character. Using wild cards can make it easier to handle related objects, provided they follow a suitable naming convention.

For example:

```
mp3*.obj    /* matches anything .obj that begins with mp3    */
task?.o*    /* matches task1.obj, task2.obj, taskX.o55, etc. */

SECTIONS
{
   .fast_code: { *.obj(*fast*) }              > FAST_MEM
   .vectors  : { vectors.obj(.vector:part1:*) > 0xFFFFFF00
   .str_code : { rts*.lib<str*.obj>(.text) }  > S1ROM
}
```

9.4.2 Relocation Capabilities (--absolute_exe and --relocatable 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 (--absolute_exe and --relocatable) that allow you to produce an absolute or a relocatable output module. The linker also supports a third option (-ar) that allows you to produce an executable, relocatable output module.

When the linker encounters a file that contains no relocation or symbol table information, it issues a warning message (but continues executing). Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated (that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

9.4.2.1 Producing an Absolute Output Module (--absolute_exe option)

When you use the --absolute_exe option without the --relocatable option, the linker produces an *absolute, executable* output module. Absolute files contain *no* relocation information. Executable files contain the following:

- Special symbols defined by the linker (see Section 9.6.9.4)
- An optional header that describes information such as the program entry point
- *No* unresolved references

The following example links file1.obj and file2.obj and creates an absolute output module called a.out:

```
cl55 --run_linker --absolute_exe file1.obj file2.obj
```

The --absolute_exe and --relocatable Options

NOTE: If you do not use the --absolute_exe or the --relocatable option, the linker acts as if you specified --absolute_exe.

9.4.2.2 Producing a Relocatable Output Module (--relocatable option)

When you use the --relocatable option, the linker retains relocation entries in the output module. If the output module is relocated (at load time) or relinked (by another linker execution), use --relocatable to retain the relocation entries.

The linker produces a file that is not executable when you use the --relocatable option without the --absolute_exe option. A file that is not executable does not contain special linker symbols or an optional header. The file can contain unresolved references, but these references do not prevent creation of an output module.

This example links file1.obj and file2.obj and creates a relocatable output module called a.out:

```
cl55 --run_linker --relocatable 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 9.10.)

9.4.2.3 Producing an Executable, Relocatable Output Module (-ar Option)

If you invoke the linker with both the --absolute_exe and --relocatable options, the linker produces an *executable, relocatable* object module. The output file contains the special linker symbols, an optional header, and all resolved symbol references; however, the relocation information is retained.

This example links file1.obj and file2.obj and creates an executable, relocatable output module called xr.out:

```
cl55 --run_linker -ar file1.obj file2.obj --output_file=xr.out
```

9.4.3 Allocate Memory for Use by the Loader to Pass Arguments (--arg_size Option)

The --arg_size option instructs the linker to allocate memory to be used by the loader to pass arguments from the command line of the loader to the program. The syntax of the --arg_size option is:

--arg_size= *size*

The *size* is a number representing the number of bytes to be allocated in target memory for command-line arguments.

By default, the linker creates the __c_args__ symbol and sets it to -1. When you specify --arg_size=*size*, the following occur:

- The linker creates an uninitialized section named .args of *size* bytes.
- The __c_args__ symbol contains the address of the .args section.

The loader and the target boot code use the .args section and the __c_args__ symbol to determine whether and how to pass arguments from the host to the target program. See the *TMS320C55x Optimizing C/C++ Compiler User's Guide* for information about the loader.

9.4.4 Control Linker Diagnostics

The linker uses certain C/C++ compiler options to control linker-generated diagnostics. The diagnostic options must be specified before the --run_linker option.

--diag_error=*num* Categorizes the diagnostic identified by *num* as an error. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_error=*num* to recategorize the diagnostic as an error. You can only alter the severity of discretionary diagnostics.

--diag_remark=*num* Categorizes the diagnostic identified by *num* as a remark. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_remark=*num* to recategorize the diagnostic as a remark. You can only alter the severity of discretionary diagnostics.

--diag_suppress=*num* Suppresses the diagnostic identified by *num*. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_suppress=*num* to suppress the diagnostic. You can only suppress discretionary diagnostics.

--diag_warning=*num* Categorizes the diagnostic identified by *num* as a warning. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_warning=*num* to recategorize the diagnostic as a warning. You can only alter the severity of discretionary diagnostics.

--display_error_number Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See the *TMS320C55x Optimizing C/C++ Compiler User's Guide* for more information on understanding diagnostic messages.

--emit_warnings_as_errors Treats all warnings as errors. This option cannot be used with the --no_warnings option. The --diag_remark option takes precedence over this option. This option takes precedence over the --diag_warning option.

--issue_remarks Issues remarks (nonserious warnings), which are suppressed by default.

--no_warnings Suppresses warning diagnostics (errors are still issued).

--set_error_limit=*num* Sets the error limit to *num*, which can be any decimal value. The linker abandons linking after this number of errors. (The default is 100.)

--verbose_diagnostics Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line

9.4.5 Disable Automatic Library Selection (--disable_auto_rts Option)

The --disable_auto_rts option disables the automatic selection of a run-time-support library. See the *TMS320C55x Optimizing C/C++ Compiler User's Guide* for details on the automatic selection process.

9.4.6 Disable Conditional Linking (--disable_clink Option)

The --disable_clink option disables removal of unreferenced sections in COFF object modules. Only sections marked as candidates for removal with the .clink assembler directive are affected by conditional linking. See Conditionally Leave Section Out of Object Module Output for details on setting up conditional linking using the .clink directive.

9.4.7 Link Command File Preprocessing (--disable_pp, --define and --undefine Options)

The linker preprocesses link command files using a standard C preprocessor. Therefore, the command files can contain well-known preprocessing directives such as #define, #include, and #if / #endif.

Three linker options control the preprocessor:

--disable_pp	Disables preprocessing for command files
--define=*name*[=*val*]	Predefines *name* as a preprocessor macro
--undefine=*name*	Removes the macro *name*

The compiler has --define and --undefine options with the same meanings. However, the linker options are distinct; only --define and --undefine options specified after --run_linker are passed to the linker. For example:

```
cl55 --define=FOO=1 main.c --run_linker --define=BAR=2 lnk.cmd
```

The linker sees only the --define for BAR; the compiler only sees the --define for FOO.

When one command file #includes another, preprocessing context is carried from parent to child in the usual way (that is, macros defined in the parent are visible in the child). However, when a command file is invoked other than through #include, either on the command line or by the typical way of being named in another command file, preprocessing context is **not** carried into the nested file. The exception to this is --define and --undefine options, which apply globally from the point they are encountered. For example:

```
--define GLOBAL
#define LOCAL

#include "incfile.cmd"     /* sees GLOBAL and LOCAL */
nestfile.cmd               /* only sees GLOBAL      */
```

Two cautions apply to the use of --define and --undefine in command files. First, they have global effect as mentioned above. Second, since they are not actually preprocessing directives themselves, they are subject to macro substitution, probably with unintended consequences. This effect can be defeated by quoting the symbol name. For example:

```
--define MYSYM=123
--undefine MYSYM     /* expands to --undefine 123 (!) */
--undefine "MYSYM"   /* ahh, that's better            */
```

The linker uses the same search paths to find #include files as it does to find libraries. That is, #include files are searched in the following places:

1. If the #include file name is in quotes (rather than <brackets>), in the directory of the current file
2. In the list of directories specified with --library options or environment variables (see Section 9.4.12)

There are two exceptions: relative pathnames (such as "../name") always search the current directory; and absolute pathnames (such as "/usr/tools/name") bypass search paths entirely.

The linker has the standard built-in definitions for the macros __FILE__, __DATE__, and __TIME__. It does not, however, have the compiler-specific options for the target (__.TMS320C55X__), version (__TI_COMPILER_VERSION__), run-time model, and so on.

9.4.8 Define an Entry Point (--entry_point Option)

The memory address at which a program begins executing is called the *entry point*. When a loader loads a program into target memory, the program counter (PC) must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table.

- The value specified by the --entry_point option. The syntax is:

  --entry_point= *global_symbol*

  where *global_symbol* defines the entry point and must be defined as an external symbol of the input files. The external symbol name of C or C++ objects may be different than the name as declared in the source language; refer to the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.
- The value of symbol _c_int00 (if present). The _c_int00 symbol *must* be the entry point if you are linking code produced by the C compiler.
- The value of symbol _main (if present)
- 0 (default value)

This example links file1.obj and file2.obj. The symbol begin is the entry point; begin must be defined as external in file1 or file2.

```
cl55 --run_linker --entry_point=begin file1.obj file2.obj
```

9.4.9 Set Default Fill Value (--fill_value Option)

The --fill_value option fills the holes formed within output sections. The syntax for the option is:

--fill_value= *value*

The argument *value* is a 32-bit constant (up to eight hexadecimal digits). If you do not use --fill_value, the linker uses 0 as the default fill value.

This example fills holes with the hexadecimal value ABCDABCD:

```
cl55 --run_linker --fill_value=0xABCDABCD file1.obj file2.obj
```

9.4.10 Define Heap Size (--heap_size Option)

The C/C++ compiler uses an uninitialized section called .sysmem for the C run-time memory pool used by malloc(). You can set the size of this memory pool at link time by using the --heap_size option. The syntax for the --heap_size option is:

--heap_size= *size*

The *size* must be a constant. This example defines a 4K byte heap:

```
cl55 --run_linker --heap_size=0x1000 /* defines a 4k heap (.sysmem section)*/
```

The linker creates the .sysmem section only if there is a .sysmem section in an input file.

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 2K bytes.

For more information about C/C++ linking, see Section 9.11.

9.4.11 Hiding Symbols

Symbol hiding prevents the symbol from being listed in the output file's symbol table. While localization is used to prevent name space clashes in a link unit, symbol hiding is used to obscure symbols which should not be visible outside a link unit. Such symbol's names appear only as empty strings or "no name" in object file readers. The linker supports symbol hiding through the --hide and --unhide options.

The syntax for these options are:

--hide=' *pattern* **'**

--unhide=' *pattern* **'**

The *pattern* is a string with optional wildcards ? or *. Use ? to match a single character and use * to match zero or more characters.

The --hide option hides global symbols which have a linkname matching the *pattern*. It hides the symbols matching the pattern by changing the name to an empty string. A global symbol which is hidden is also localized.

The --unhide option reveals (un-hides) global symbols that match the *pattern* that are hidden by the --hide option. The --unhide option excludes symbols that match pattern from symbol hiding provided the pattern defined by --unhide is more restrictive than the pattern defined by --hide.

These options have the following properties:

- The --hide and --unhide options can be specified more than once on the command line.
- The order of --hide and --unhide has no significance.
- A symbol is matched by only one pattern defined by either --hide or --unhide.
- A symbol is matched by the most restrictive pattern. Pattern A is considered more restrictive than Pattern B, if Pattern A matches a narrower set than Pattern B.
- It is an error if a symbol matches patterns from --hide and --unhide and if one does not supersede other. Pattern A supersedes pattern B if A can match everything B can, and some more. If Pattern A supersedes Pattern B, then Pattern B is said to more restrictive than Pattern A.
- These options affect final and partial linking.

In map files these symbols are listed under the Hidden Symbols heading.

9.4.12 Alter the Library Search Algorithm (--library Option, --search_path Option, and C55X_C_DIR Environment Variable)

Usually, when you want to specify a file as linker input, you simply enter the filename; the linker looks for the file in the current directory. For example, suppose the current directory contains the library object.lib. Assume that this library defines symbols that are referenced in the file file1.obj. This is how you link the files:

```
cl55 --run_linker file1.obj object.lib
```

If you want to use a file that is not in the current directory, use the --library linker option. The --library option's short form is -l. The syntax for this option is:

--library=[*pathname*] *filename*

The *filename* is the name of an archive, an object file, or link command file. You can specify up to 128 search paths.

The --library option is not required when one or more members of an object library are specified for input to an output section. For more information about allocating archive members, see Section 9.6.4.5.

You can augment the linker's directory search algorithm by using the --search_path linker option or the C55X_C_DIR environment variable. The linker searches for object libraries and command files in the following order:

1. It searches directories named with the --search_path linker option. The --search_path option must appear before the --library option on the command line or in a command file.
2. It searches directories named with C55X_C_DIR.
3. If C55X_C_DIR is not set, it searches directories named with the assembler's C55X_A_DIR environment variable.
4. It searches the current directory.

9.4.12.1 Name an Alternate Library Directory (--search_path Option)

The --search_path option names an alternate directory that contains input files. The --search_path option's short form is -I. The syntax for this option is:

--search_path= *pathname*

The *pathname* names a directory that contains input files.

When the linker is searching for input files named with the --library option, it searches through directories named with --search_path first. Each --search_path option specifies only one directory, but you can have several --search_path options per invocation. When you use the --search_path option to name an alternate directory, it must precede any --library option used on the command line or in a command file.

For example, assume that there are two archive libraries called r.lib and lib2.lib that reside in ld and ld2 directories. 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	Enter
UNIX (Bourne shell)	`cl55 --run_linker f1.obj f2.obj --search_path=/ld --search_path=/ld2 --library=r.lib --library=lib2.lib`
Windows	`cl55 --run_linker f1.obj f2.obj --search_path=\ld --search_path=\ld2 --library=r.lib --library=lib2.lib`

9.4.12.2 Name an Alternate Library Directory (C55X_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 C55X_C_DIR to name alternate directories that contain object libraries. The command syntaxes for assigning the environment variable are:

Operating System	Enter
UNIX (Bourne shell)	**C55X_C_DIR="** *pathname*$_1$**;** *pathname*$_2$**; . . . "; export C55X_C_DIR**
Windows	**set C55X_C_DIR=** *pathname*$_1$ **;** *pathname*$_2$ **; . . .**

The *pathnames* are directories that contain input files. Use the --library linker option on the command line or in a command file to tell the linker which library or link command file to search for. The pathnames must follow these constraints:

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

  ```
  set C55X_C_DIR= c:\path\one\to\tools ; c:\path\two\to\tools
  ```

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

  ```
  set C55X_C_DIR=c:\first path\to\tools;d:\second path\to\tools
  ```

In the example below, assume that two archive libraries called r.lib and lib2.lib reside in ld and ld2 directories. The table below shows how to set the environment variable, and how to use both libraries during a link. Select the row for your operating system:

Operating System	Invocation Command
UNIX (Bourne shell)	`C55X_C_DIR="/ld ;/ld2"; export C55X_C_DIR;` `cl55 --run_linker f1.obj f2.obj --library=r.lib --library=lib2.lib`
Windows	`C55X_C_DIR=\ld;\ld2` `cl55 --run linker f1.obj f2.obj --library=r.lib --library=lib2.lib`

The environment variable remains set until you reboot the system or reset the variable by entering:

Operating System	Enter
UNIX (Bourne shell)	`unset C55X_C_DIR`
Windows	`set C55X_C_DIR=`

The assembler uses an environment variable named C55X_A_DIR to name alternate directories that contain copy/include files or macro libraries. If C55X_C_DIR is not set, the linker searches for object libraries in the directories named with C55X_A_DIR. For information about C55X_A_DIR, see Section 3.5.2. For more information about object libraries, see Section 9.7.

9.4.12.3 Exhaustively Read and Search Libraries (--reread_libs and --priority Options)

There are two ways to exhaustively search for unresolved symbols:

- Reread libraries if you cannot resolve a symbol reference (--reread_libs).
- Search libraries in the order that they are specified (--priority).

The linker normally reads input files, including archive libraries, only once when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library, the reference is not resolved.

With the --reread_libs option, you can force the linker to reread all libraries. The linker rereads libraries until no more references can be resolved. Linking using --reread_libs 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:

```
cl55 --run_linker --library=a.lib --library=b.lib --library=a.lib
```

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

```
cl55 --run_linker --reread_libs --library=a.lib --library=b.lib
```

The --priority option provides an alternate search mechanism for libraries. Using --priority causes each unresolved reference to be satisfied by the first library that contains a definition for that symbol. For example:

```
objfile  references A
lib1     defines B
lib2     defines A, B; obj defining A references B

% cl55 --run_linker objfile lib1 lib2
```

Under the existing model, objfile resolves its reference to A in lib2, pulling in a reference to B, which resolves to the B in lib2.

Under --priority, objfile resolves its reference to A in lib2, pulling in a reference to B, but now B is resolved by searching the libraries in order and resolves B to the first definition it finds, namely the one in lib1.

The --priority option is useful for libraries that provide overriding definitions for related sets of functions in other libraries without having to provide a complete version of the whole library.

For example, suppose you want to override versions of malloc and free defined in the rts55.lib without providing a full replacement for rts55.lib. Using --priority and linking your new library before rts55.lib guarantees that all references to malloc and free resolve to the new library.

The --priority option is intended to support linking programs with SYS/BIOS where situations like the one illustrated above occur.

9.4.13 Change Symbol Localization

Symbol localization changes symbol linkage from global to local (static). This is used to obscure global symbols in a library which should not be visible outside the library, but must be global because they are accessed by several modules in the library. The linker supports symbol localization through the --localize and --globalize linker options.

The syntax for these options are:

--localize=' *pattern* **'**

--globalize=' *pattern* **'**

The *pattern* is a string with optional wild cards ? or *. Use ? to match a single character and use * to match zero or more characters.

The --localize option changes the symbol linkage to local for symbols matching the *pattern*.

The --globalize option changes the symbol linkage to global for symbols matching the *pattern*. The --globalize option only affects symbols that are localized by the --localize option. The --globalize option excludes symbols that match the pattern from symbol localization, provided the pattern defined by --globalize is more restrictive than the pattern defined by --localize.

Specifying C/C++ Symbols with --localize and --globalize

NOTE: For COFF ABI, the compiler prepends an underscore _ to the beginning of all C/C++ identifiers. That is, for a function named foo2(), foo2() is prefixed with _ and _foo2 becomes the link-time symbol. The --localize and --globalize options accept the link-time symbols. Thus, you specify --localize='_foo2' to localize the C function _foo2(). For more information on linknames see the C/C++ Language Implementation chapter in the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.

These options have the following properties:

- The --localize and --globalize options can be specified more than once on the command line.
- The order of --localize and --globalize options has no significance.
- A symbol is matched by only one pattern defined by either --localize or --globalize.
- A symbol is matched by the most restrictive pattern. Pattern A is considered more restrictive than Pattern B, if Pattern A matches a narrower set than Pattern B.
- It is an error if a symbol matches patterns from --localize and --globalize and if one does not supersede other. Pattern A supersedes pattern B if A can match everything B can, and some more. If Pattern A supersedes Pattern B, then Pattern B is said to more restrictive than Pattern A.
- These options affect final and partial linking.

In map files these symbols are listed under the Localized Symbols heading.

9.4.13.1 Make All Global Symbols Static (--make_static Option)

The --make_static option makes all global symbols static. Static symbols are not visible to externally linked modules. By making global symbols static, global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.

The --make_static option effectively nullifies all .global assembler directives. All symbols become local to the module in which they are defined, so no external references are possible. For example, assume file1.obj and file2.obj both define global symbols called EXT. By using the --make_static option, you can link these files without conflict. The symbol EXT defined in file1.obj is treated separately from the symbol EXT defined in file2.obj.

```
cl55 --run_linker --make_static file1.obj file2.obj
```

9.4.13.2 Make a Symbol Global (--make_global Option)

The --make_static option makes all global symbols static. If you have a symbol that you want to remain global and you use the --make_static option, you can use the --make_global option to declare that symbol to be global. The --make_global option overrides the effect of the --make_static option for the symbol that you specify. The syntax for the --make_global option is:

--make_global= *global_symbol*

9.4.14 Create a Map File (--map_file Option)

The syntax for the --map_file option is:

--map_file= *filename*

The linker map describes:

- Memory configuration
- Input and output section allocation
- Linker-generated copy tables
- The addresses of external symbols after they have been relocated
- Hidden and localized symbols

The map file contains the name of the output module and the entry point; it can also contain up to three tables:

- A table showing the new memory configuration if any nondefault memory is specified (memory configuration). The table has the following columns; this information is generated on the basis of the information in the MEMORY directive in the link command file:
  - **Name.** This is the name of the memory range specified with the MEMORY directive.
  - **Origin.** This specifies the starting address of a memory range.
  - **Length.** This specifies the length of a memory range.
  - **Unused.** This specifies the total amount of unused (available) memory in that memory area.
  - **Attributes.** This specifies one to four attributes associated with the named range:

    R specifies that the memory can be read.
    W specifies that the memory can be written to.
    X specifies that the memory can contain executable code.
    I specifies that the memory can be initialized.

  For more information about the MEMORY directive, see Section 9.6.3.
- A table showing the linked addresses of each output section and the input sections that make up the output sections (section allocation map). This table has the following columns; this information is generated on the basis of the information in the SECTIONS directive in the link command file:
  - **Output section.** This is the name of the output section specified with the SECTIONS directive.
  - **Origin.** The first origin listed for each output section is the starting address of that output section. The indented origin value is the starting address of that portion of the output section.
  - **Length.** The first length listed for each output section is the length of that output section. The indented length value is the length of that portion of the output section.
  - **Attributes/input sections.** This lists the input file or value associated with an output section. If the input section could not be allocated, the map file will indicate this with "FAILED TO ALLOCATE".

  For more information about the SECTIONS directive, see Section 9.6.4.
- A table showing each external symbol and its address sorted by symbol name.
- A table showing each external symbol and its address sorted by symbol address.

The following example links file1.obj and file2.obj and creates a map file called map.out:

```
cl55 --run_linker file1.obj file2.obj --map_file=map.out
```

Example 9-28 shows an example of a map file.

9.4.15 Managing Map File Contents (--mapfile_contents Option)

The --mapfile_contents option assists with managing the content of linker-generated map files. The syntax for the --mapfile_contents option is:

--mapfile_contents= *filter*[, *filter*]

When the --map_file option is specified, the linker produces a map file containing information about memory usage, placement information about sections that were created during a link, details about linker-generated copy tables, and symbol values.

The new --mapfile_contents option provides a mechanism for you to control what information is included in or excluded from a map file. When you specify --mapfile_contents=help from the command line, a help screen listing available filter options is displayed.

The following filter options are available:

Attribute	Description	Default State
copytables	Copy tables	On
entry	Entry point	On
load_addr	Display load addresses	Off
memory	Memory ranges	On
sections	Sections	On
sym_defs	Defined symbols per file	Off
sym_name	Symbols sorted by name	On
sym_runaddr	Symbols sorted by run address	On
all	Enables all attributes	
none	Disables all attributes	

The --mapfile_contents option controls display filter settings by specifying a comma-delimited list of display attributes. When prefixed with the word no, an attribute is disabled instead of enabled. For example:

```
--mapfile_contents=copytables,noentry
--mapfile_contents=all,nocopytables
--mapfile_contents=none,entry
```

By default, those sections that are currently included in the map file when the --map_file option is specified are included. The filters specified in the --mapfile_contents options are processed in the order that they appear in the command line. In the third example above, the first filter, none, clears all map file content. The second filter, entry, then enables information about entry points to be included in the generated map file. That is, when --mapfile_contents=none,entry is specified, the map file contains *only* information about entry points.

There are two new filters included with the --mapfile_contents option, load_addr and sym_defs. These are both disabled by default. If you turn on the load_addr filter, the map file includes the load address of symbols that are included in the symbol list in addition to the run address (if the load address is different from the run address).

The sym_defs filter can be used to include information about all static and global symbols defined in an application on a file by file basis. You may find it useful to replace the sym_name and sym_runaddr sections of the map file with the sym_defs section by specifying the following --mapfile_contents option:

```
--mapfile_contents=nosym_name,nosym_runaddr,sym_defs
```

9.4.16 Disable Name Demangling (--no_demangle)

By default, the linker uses demangled symbol names in diagnostics. For example:

```
undefined symbol                first referenced in file
ANewClass::getValue()           test.obj
```

The --no_demangle option disables the demangling of symbol names in diagnostics. For example:

```
undefined symbol                first referenced in file
_ZN9ANewClass8getValueEv        test.obj
```

9.4.17 Disable Merge of Symbolic Debugging Information (--no_sym_merge Option)

By default, the linker eliminates duplicate entries of symbolic debugging information. Such duplicate information is commonly generated when a C program is compiled for debugging. For example:

```
-[ header.h ]-
typedef struct
{
   <define some structure members>
} XYZ;

-[ f1.c ]-
#include "header.h"
...

-[ f2.c ]-
#include "header.h"
...
```

When these files are compiled for debugging, both f1.obj and f2.obj have symbolic debugging entries to describe type XYZ. For the final output file, only one set of these entries is necessary. The linker eliminates the duplicate entries automatically.

Use the COFF only --no_sym_merge option if you want the linker to keep such duplicate entries in COFF object files. Using the --no_sym_merge option has the effect of the linker running faster and using less host memory during linking, but the resulting executable file may be very large due to duplicated debug information.

9.4.18 Strip Symbolic Information (--no_symtable Option)

The --no_symtable option creates a smaller output module by omitting symbol table information and line number entries. The --no_sym_table option is useful for production applications when you do not want to disclose symbolic information to the consumer.

This example links file1.obj and file2.obj and creates an output module, stripped of line numbers and symbol table information, named nosym.out:

```
cl55 --run_linker --output_file=nosym.out --no_symtable file1.obj file2.obj
```

Using the --no_symtable option limits later use of a symbolic debugger.

Stripping Symbolic Information

NOTE: The --no_symtable option is deprecated. To remove symbol table information, use the strip55 utility as described in Section 12.4.

9.4.19 Name an Output Module (--output_file 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 --output_file option. The syntax for the --output_file option is:

--output_file= *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:

```
cl55 --run_linker --output_file=run.out file1.obj file2.obj
```

9.4.20 C Language Options (--ram_model and --rom_model Options)

The --ram_model and --rom_model options cause the linker to use linking conventions that are required by the C compiler.

- The --ram_model option tells the linker to initialize variables at load time.
- The --rom_model option tells the linker to autoinitialize variables at run time.

For more information, see Section 9.11, Section 9.11.4, and Section 9.11.5.

9.4.21 Create an Absolute Listing File (--run_abs Option)

The --run_abs option produces an output file for each file that was linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

9.4.22 Scan All Libraries for Duplicate Symbol Definitions (--scan_libraries)

The --scan_libraries option scans all libraries during a link looking for duplicate symbol definitions to those symbols that are actually included in the link. The scan does not consider absolute symbols or symbols defined in COMDAT sections. The --scan_libraries option helps determine those symbols that were actually chosen by the linker over other existing definitions of the same symbol in a library.

The library scanning feature can be used to check against unintended resolution of a symbol reference to a definition when multiple definitions are available in the libraries.

9.4.23 Define Stack Size (--stack_size Option)

The TMS320C55x C/C++ compiler uses an uninitialized section, .stack, to allocate space for the run-time stack. You can set the size of this section in bytes at link time with the --stack_size option. The syntax for the --stack_size option is:

--stack_size= *size*

The *size* must be a constant and is in bytes. This example defines a 4K byte stack:

```
cl55 --run_linker --stack_size=0x1000 /* defines a 4K heap (.stack section)*/
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size is different.

When the linker defines the .stack section, it also defines a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the section. The default software stack size is 1K bytes.

Allocation of .stack and .sysstack Sections

NOTE: The .stack and .sysstack sections must be allocated on the same 64K-word data page.

9.4.24 Enforce Strict Compatibility (--strict_compatibility Option)

The linker performs more conservative and rigorous compatibility checking of input object files when you specify the --strict_compatibility option. Using this option guards against additional potential compatibility issues, but may signal false compatibility errors when linking in object files built with an older toolset, or with object files built with another compiler vendor's toolset. To avoid issues with legacy libraries, the --strict_compatibility option is turned off by default.

9.4.25 Mapping of Symbols (--symbol_map Option)

Symbol mapping allows a symbol reference to be resolved by a symbol with a different name. Symbol mapping allows functions to be overridden with alternate definitions. This feature can be used to patch in alternate implementations, which provide patches (bug fixes) or alternate functionality. The syntax for the --symbol_map option is:

--symbol_map= *refname=defname*

For example, the following code makes the linker resolve any references to foo by the definition foo_patch:

```
--symbol_map='foo=foo_patch'
```

9.4.26 Define Secondary Stack Size (--sysstack Option)

The TMS320C55x C/C++ compiler uses an uninitialized section, .sysstack, to allocate space for the secondary run-time stack. You can set the size of this section in bytes at link time with the --sysstack option. The syntax for the --sysstack option is:

--sysstack= *size*

The *size* must be a constant and is in bytes. This example defines a 4K byte stack:

```
--sysstack=0x1000 /* defines the secondary stack size) */
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size is different.

When the linker defines the .sysstack section, it also defines a global symbol, __SYSSTACK_SIZE, and assigns it a value equal to the size of the section. The default software stack size is 1K bytes.

9.4.27 Introduce an Unresolved Symbol (--undef_sym Option)

The --undef_sym option introduces the linkname for 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 --undef_sym option *before* it links in the member that defines the symbol. The syntax for the --undef_sym option is:

--undef_sym= *symbol*

For example, suppose a library named rts55.lib contains a member that defines the symbol symtab; none of the object files being linked reference symtab. However, suppose you plan to relink the output module and you want to include the library member that defines symtab in this link. Using the --undef_sym option as shown below forces the linker to search rts55.lib for the member that defines symtab and to link in the member.

```
cl55 --run_linker --undef_sym=symtab file1.obj file2.obj rts55.lib
```

If you do not use --undef_sym, this member is not included, because there is no explicit reference to it in file1.obj or file2.obj.

9.4.28 Display a Message When an Undefined Output Section Is Created (--warn_sections Option)

The --warn_sections option displays additional messages pertaining to the default creation of output sections. Additional messages are displayed in the following circumstances:

- In a link command file, you can set up a SECTIONS directive that describes how input sections are combined into output sections. However, if the linker encounters one or more input sections that do not have a corresponding output section defined in the SECTIONS directive, the linker combines the input sections that have the same name into an output section with that name. By default, the linker does not display a message to tell you that this occurred.

  If this situation occurs and you use the --warn_section option, the linker displays a message when it creates a new output section.
- If you do not use the --heap, --stack_size, and --sysstack options, the linker creates the .sysmem, .stack, and .sysstack (respectively) sections for you. The .sysmem section has a default size of 2K bytes; the .stack and .sysstack sections have a default size of 1K bytes. You might not have enough memory available for one or all of these sections. In this case, the linker issues an error message saying a section could not be allocated.

  If you use the --warn_section option, the linker displays another message with more details, which includes the name of the directive to allocate the .sysmem or .stack section yourself.

Allocation of .stack and .sysstack Sections

NOTE: The .stack and .sysstack sections must be allocated on the same 64K-word data page.

For more information about the SECTIONS directive, see Section 9.6.4. For more information about the default actions of the linker, see Section 9.8.

9.4.29 Generate XML Link Information File (--xml_link_info Option)

The linker supports the generation of an XML link information file through the --xml_link_info=*file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker generated map file.

See Appendix B for specifics on the contents of the generated XML file.

9.5 Byte/Word Addressing

C55x memory is byte-addressable for code and word-addressable for data. The assembler and linker keep track of the addresses, relative offsets, and sizes of the bits in units that are appropriate for the given section: words for data sections, and bytes for code sections.

Use Byte Addresses in Linker Command File

NOTE: All addresses and sizes supplied in the linker command file should be byte addresses, for both code and data sections.

In the case of program labels, the unchanged byte addresses are encoded in the executable output and during execution sent over the program address bus. In the case of data labels, the byte addresses are divided by 2 in the linker (converting them to word addresses) prior to being encoded in the executable output and sent over the data address bus.

The .map file created by the linker shows code addresses and sizes in bytes, and data addresses and sizes in words.

9.6 Linker Command Files

Linker command files allow you to put linking information in a file; this is useful when you invoke the linker often with the same information. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line.

Linker command files are ASCII files that contain one or more of the following:

- Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)
- Linker options, which can be used in the command file in the same manner that they are used on the command line
- The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration (see Section 9.6.3). The SECTIONS directive controls how sections are built and allocated (see Section 9.6.4.)
- Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the cl55 --run_linker command and follow it with the name of the command file:

cl55 --run_linker *command_filename*

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links the file. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it. Command filenames are case sensitive, regardless of the system used.

Example 9-1 shows a sample link command file called link.cmd.

Example 9-1. Linker Command File

```
a.obj                    /*  First input filename          */
b.obj                    /*  Second input filename         */
--output_file=prog.out   /*  Option to specify output file */
--map_file=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:

```
cl55 --run_linker link.cmd
```

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

```
cl55 --run_linker --relocatable link.cmd c.obj d.obj
```

The linker processes the command file as soon as it encounters the filename, so a.obj and b.obj are linked into the output module before c.obj and d.obj.

You can specify multiple command files. If, for example, you have a file called names.lst that contains filenames and another file called dir.cmd that contains linker directives, you could enter:

```
cl55 --run_linker 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.

Example 9-2. Command File With Linker Directives

```
a.obj b.obj c.obj           /* Input filenames       */
--output_file=prog.out      /* Options               */
--map_file=prog.map

MEMORY                      /* MEMORY directive      */
{
  FAST_MEM:  origin = 0x0100    length = 0x0100
  SLOW_MEM:  origin = 0x1000    length = 0x0100
}

SECTIONS                    /* SECTIONS directive    */
{
  .text:  > SLOW_MEM
  .data:  > FAST_MEM
  .bss:   > FAST_MEM
}
```

For more information, see Section 9.6.3 for the MEMORY directive, and Section 9.6.4 for the SECTIONS directive.

9.6.1 Reserved Names in Linker Command Files

The following names (in lowercase also) are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

ALIGN	FILL	LOAD_SIZE	PAGE	START
ATTR	GROUP	LOAD_START	PALIGN	TABLE
BLOCK	HIGH	MEMORY	RUN	TYPE
COMPRESSION	l (lowercase L)	NOINIT	RUN_END	UNION
COPY	len	NOLOAD	RUN_SIZE	UNORDERED
DSECT	LENGTH	o	RUN_START	
END	LOAD	org	SECTIONS	
f	LOAD_END	ORIGIN	SIZE	

9.6.2 Constants in Linker Command Files

You can specify constants with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants used in the assembler (see Section 3.7) or the scheme used for integer constants in C syntax.

Examples:

Format	Decimal	Octal	Hexadecimal
Assembler format	32	40q	020h
C format	32	040	0x20

9.6.3 The MEMORY Directive

The linker determines where output sections are allocated into memory; it must have a model of target memory to accomplish this. The MEMORY directive allows you to specify a model of target memory so that you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections and uses it to determine which memory locations can be used for object code.

The memory configurations of TMS320C55x systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use MEMORY to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

For more information, see Section 2.3 and Section 2.4.

9.6.3.1 Default Memory Model

The assembler enables you to assemble code for the TMS320C55x 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. For more information about the default memory model, see Section 9.8.

9.6.3.2 MEMORY Directive Syntax

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

- Page
- Name
- Starting address
- Length
- Optional set of attributes
- Optional fill specification

By default, the linker uses a single address space on PAGE 0. However, the linker allows you to configure separate address spaces by using the MEMORY directive's PAGE option. The PAGE option causes the linker to treat the specified pages as completely separate memory spaces. C55x 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 memory ranges that are available for loading code. Memory defined by the MEMORY directive is configured; any memory that you do not explicitly account for with MEMORY is unconfigured. The linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. Example 9-3 shows a sample command file with the MEMORY directive. (For details on the start/end/size address operators see Example 9-4.).

Example 9-3. The MEMORY Directive

```
/************************************************/
/*   Sample command file with MEMORY directive    */
/************************************************/
file1.obj file2.obj              /*  Input files  */
--output_file=prog.out          /*    Options    */

MEMORY
{
      SLOW_MEM:    origin = 0x1C00, length = 0x1000

      SCRATCH:     origin = 0x0060, length = 0x0020
      FAST_MEM:    origin = 0x0080, length = 0x1000
}
```

The general syntax for the MEMORY directive is:

MEMORY

{

[**PAGE 0:**] *name 1* [**(** *attr* **)**] **: origin =** *expression* **, length =** *expression* [**, fill =** *constant*]

[**PAGE 1:**] *name 2* [**(** *attr* **)**] **: origin =** *expression* **, length =** *expression* [**, fill =** *constant*];

.

.

[**PAGE** *n*:] *name n* [**(** *attr* **)**] **: origin =** *expression* **, length =** *expression* [**, fill =** *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 2 specifies peripheral 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 2.

name names a memory range. A memory name can be one to 64 characters; valid characters include A-Z, a-z, $, ., and _. The names have no special significance to the linker; they simply identify memory ranges. Memory range names are internal to the linker and are not retained in the output file or in the symbol table. All memory ranges must have unique names and must not overlap.

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 are:

- **R** specifies that the memory can be read.
- **W** specifies that the memory can be written to.
- **X** specifies that the memory can contain executable code.
- **I** specifies that the memory can be initialized.

origin specifies the starting address of a memory range; enter as *origin*, *org*, or *o*. The value, specified in bytes, is an expression of 24-bit constants, which can 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 an expression of 24-bit constants, which can 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 can be decimal, octal, or hexadecimal. The fill value is used to fill areas of the memory range that are not allocated to a section.

Filling Memory Ranges

NOTE: If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

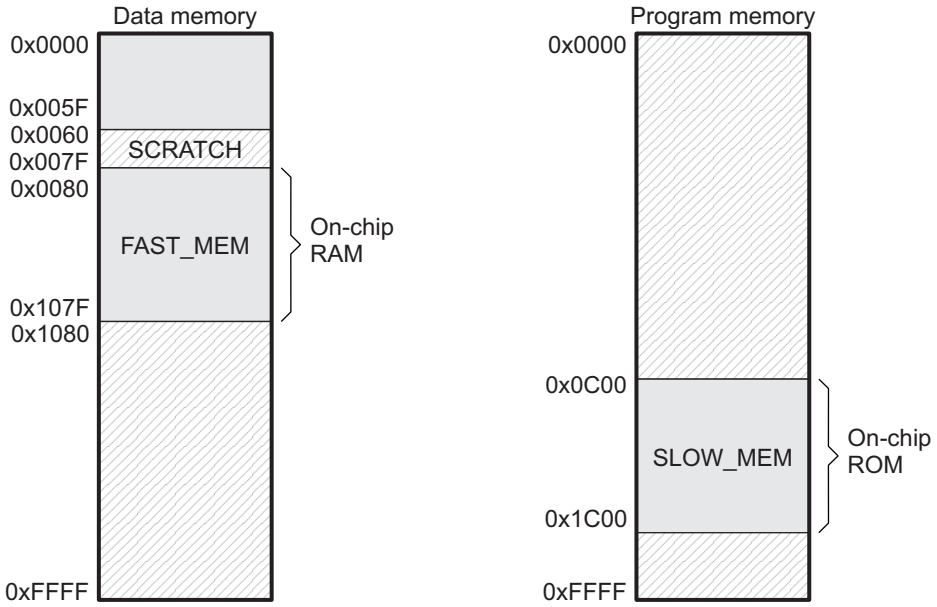
The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFh:

```
MEMORY
{
    RFILE (RW) : o = 0x0020, l = 0x00FE, f = 0x0FFF
}
```

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control allocation of output sections. After you use MEMORY to specify the target system's memory model, you can use SECTIONS to allocate output sections into specific named memory ranges or into memory that has specific attributes. For example, you could allocate the .text and .data sections into the area named FAST_MEM and allocate the .bss section into the area named SLOW_MEM.

Figure 9-2 illustrates the memory map shown in Example 9-3

Figure 9-2. Memory Map Defined in Example 9-3



9.6.3.3 Expressions and Address Operators

Memory range origin and length can now use expressions of integer constants with below operators:

Binary operators:	* / % + - << >> == != < <= > >= & | && ||
Unary operators:	- ~ !

Expressions are evaluated using standard C operator precedence rules.

No checking is done for overflow or underflow, however, expressions are evaluated using a larger integer type.

Preprocess directive #define constants can be used in place of integer constants. Global symbols cannot be used in Memory Directive expressions.

Three new address operators have been added for referencing memory range properties from prior memory range entries:

START(MR[,PAGE])	Returns start address for previously defined memory range MR.
SIZE(MR[,PAGE])	Returns size of previously defined memory range MR.
END(MR[,PAGE])	Returns end address for previously defined memory range MR.

NOTE: If no PAGE information is input then PAGE=0.

Example 9-4. Origin and Length as Expressions

```
/*******************************************************/
/*      Sample command file with MEMORY directive      */
/*******************************************************/
file1.obj file2.obj              /*    Input files     */
--output_file=prog.out           /*    Options         */
#define ORIGIN 0x00000000
#define BUFFER 0x00000200
#define CACHE  0x0001000

MEMORY
{
   FAST_MEM (RX): origin = ORIGIN + CACHE length = 0x00001000 + BUFFER
   SLOW_MEM (RW): origin = end(FAST_MEM)  length = 0x00001800 - size(FAST_MEM)
   EXT_MEM  (RX): origin = 0x10000000     length = size(FAST_MEM) - CACHE
```

9.6.4 The SECTIONS Directive

The SECTIONS directive controls your sections in the following ways:

- Describes how input sections are combined into output sections
- Defines output sections in the executable program
- Specifies where output sections are placed in memory (in relation to each other and to the entire memory space)
- Permits renaming of output sections

For more information, see Section 2.3, Section 2.4, and Section 2.2.4. Subsections allow you to manipulate sections with greater precision.

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. Section 9.8 describes this algorithm in detail.

9.6.4.1 SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

SECTIONS
{
name : [*property* [**, property**] [**, property**] . . .]
name : [*property* [**, property**] [**, property**] . . .]
name : [*property* [**, property**] [**, property**] . . .]
}

Each section specification, beginning with *name*, defines an output section. (An output section is a section in the output file.) A section name can be a subsection specification. (See Section 9.6.4.4 for information on multi-level subsections.) After the section name is a list of properties that define the section's contents and how the section is allocated. The properties can be separated by optional commas. Possible properties for a section are as follows:

- **Load allocation** defines where in memory the section is to be loaded.
  Syntax: **load =** *allocation* or
  allocation or
  > *allocation*
- **Run allocation** defines where in memory the section is to be run.
  Syntax: **run =** *allocation* or
  run > *allocation*
- **Input sections** defines the input sections (object files) that constitute the output section.
  Syntax: **{** *input_sections* **}**
- **Section type** defines flags for special section types. See Section 9.6.8
  Syntax: **type = COPY** or
  type = DSECT or
  type = NOLOAD
- **Fill value** defines the value used to fill uninitialized holes. See Section 9.6.10.
  Syntax: **fill =** *value* or
  name : [*properties* **=** *value*]

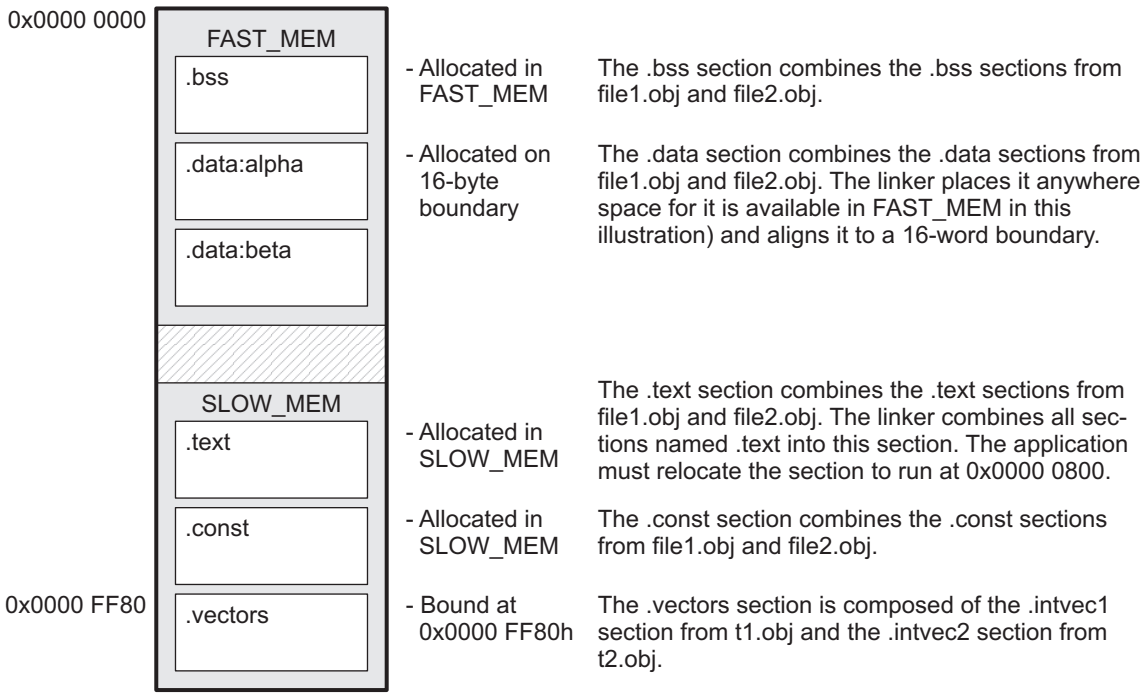
Example 9-5 shows a SECTIONS directive in a sample link command file.

Example 9-5. The SECTIONS Directive

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

SECTIONS
{
   .text:       load = EXT_MEM, run = 0x0800
   .const:      load = FAST_MEM
   .bss:        load = SLOW_MEM
   .vectors:    load = 0x0000
      {
         t1.obj(.intvec1)
         t2.obj(.intvec2)
         endvec = .;
      }
   .data:alpha: align = 16
   .data:beta:  align = 16
}
```

Figure 9-3 shows the six output sections defined by the SECTIONS directive in Example 9-5 (.vectors, .text, .const, .bss, .data:alpha, and .data:beta) and shows how these sections are allocated in memory using the MEMORY directive given in Example 9-3.

Figure 9-3. Section Allocation Defined by Example 9-5



9.6.4.2 Allocation

The linker assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. The process of locating the output section in the target's memory and assigning its address(es) is called allocation. For more information about using separate load and run allocation, see Section 9.6.5.

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to allocate the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run allocation are separate, all parameters following the keyword LOAD apply to load allocation, and those following the keyword RUN apply to run allocation. The allocation parameters are:

Binding allocates a section at a specific address.
```
.text: load = 0x1000
```

Named memory allocates the section into a range defined in the MEMORY directive with the specified name (like SLOW_MEM) or attributes.
```
.text: load > SLOW_MEM
```

Alignment uses the align or palign keyword to specify that the section must start on an address boundary.
```
.text: align = 0x100
```
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 byte boundary:
```
SECTIONS
 {
    outsect: { bar.obj(.bss)
                    . = align(0x40);
                }
 }
```

Splitting uses the split operator to list memory areas in which the section can be placed.
```
.text: >> ROM1|ROM2|ROM3
```

Blocking uses the block keyword to specify that the section must fit between two address boundaries: if the section is too big, it starts on an address boundary.
```
.text: block(0x100)
```

Page specifies the memory page to be used (see Section 9.6.7). If PAGE is not specified, the linker allocates initialized sections to PAGE 0 (program memory) and uninitialized sections to PAGE 1 (data memory).
```
.text: PAGE 0
```

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

```
.text: > SLOW_MEM
.text: {...} > SLOW_MEM
.text: > 0x4000
```

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

```
.text: > SLOW_MEM align 16 PAGE 2
```

Or if you prefer, use parentheses for readability:

```
.text: load = (SLOW_MEM align(16)) page 2
```

You can also use an input section specification to identify the sections from input files that are combined to form an output section. See Section 9.6.4.3.

9.6.4.2.1 Binding

You can supply a specific starting address for an output section by following the section name with an address:

```
.text: 0x00001000
```

This example specifies that the .text section must begin at location 0x1000. The binding address must be a 24-bit constant.

Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

Binding is Incompatible With Alignment and Named Memory

NOTE: You cannot bind a section to an address if you use alignment or named memory. If you try to do this, the linker issues an error message.

9.6.4.2.2 Named Memory

You can allocate a section into a memory range that is defined by the MEMORY directive (see Section 9.6.3). This example names ranges and links sections into them:

```
MEMORY
{
   SLOW_MEM (RIX)  : origin = 0x00000000,  length = 0x00001000
   FAST_MEM (RWIX) : origin = 0x03000000,  length = 0x00000300
}

SECTIONS
{
   .text  :   > SLOW_MEM
   .data  :   > FAST_MEM ALIGN(128)
   .bss   :   > FAST_MEM
}
```

In this example, the linker places .text into the area called SLOW_MEM. The .data and .bss output sections are allocated into FAST_MEM. You can align a section within a named memory range; the .data section is aligned on a 128-byte boundary within the FAST_MEM range.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
   .text: > (X)   /* .text --> executable memory      */
   .data: > (RI)  /* .data --> read or init memory    */
   .bss : > (RW)  /* .bss --> read or write memory    */
}
```

In this example, the .text output section can be linked into either the SLOW_MEM or FAST_MEM area because both areas have the X attribute. The .data section can also go into either SLOW_MEM or FAST_MEM because both areas have the R and I attributes. The .bss output section, however, must go into the FAST_MEM area because only FAST_MEM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no conflicting assignments exist, the .text section starts at address 0. If a section must start on a specific address, use binding instead of named memory.

9.6.4.2.3 Controlling Allocation Using The HIGH Location Specifier

The linker allocates output sections from low to high addresses within a designated memory range by default. Alternatively, you can cause the linker to allocate a section from high to low addresses within a memory range by using the HIGH location specifier in the SECTION directive declaration.

For example, given this MEMORY directive:

```
MEMORY
{
    RAM             : origin = 0x0200, length = 0x0800
    FLASH           : origin = 0x1100, length = 0xEEE0
    VECTORS         : origin = 0xFFE0, length = 0x001E
    RESET           : origin = 0xFFFE, length = 0x0002
}
```

and an accompanying SECTIONS directive:

```
SECTIONS
{
    .bss     : {} > RAM
    .sysmem  : {} > RAM
    .stack   : {} > RAM (HIGH)
}
```

The HIGH specifier used on the .stack section allocation causes the linker to attempt to allocate .stack into the higher addresses within the RAM memory range. The .bss and .sysmem sections are allocated into the lower addresses within RAM. Example 9-6 illustrates a portion of a map file that shows where the given sections are allocated within RAM for a typical program.

Example 9-6. Linker Allocation With the HIGH Specifier

```
.bss       0    00000200    00000270     UNINITIALIZED
                  00000200    0000011a     rtsxxx.lib  : defs.obj (.bss)
                  0000031a    00000088                 : trgdrv.obj (.bss)
                  000003a2    00000078                 : lowlev.obj (.bss)
                  0000041a    00000046                 : exit.obj (.bss)
                  00000460    00000008                 : memory.obj (.bss)
                  00000468    00000004                 : _lock.obj (.bss)
                  0000046c    00000002                 : fopen.obj (.bss)
                  0000046e    00000002     hello.obj (.bss)

.sysmem    0    00000470    00000120     UNINITIALIZED
                  00000470    00000004     rtsxxx .lib : memory.obj (.sysmem)

.stack     0    000008c0    00000140     UNINITIALIZED
                  000008c0    00000002     rtsxxx .lib : boot.obj (.stack)
```

As shown in Example 9-6 , the .bss and .sysmem sections are allocated at the lower addresses of RAM (0x0200 - 0x0590) and the .stack section is allocated at address 0x08c0, even though lower addresses are available.

Without using the HIGH specifier, the linker allocation would result in the code shown in Example 9-7

The HIGH specifier is ignored if it is used with specific address binding or automatic section splitting (>> operator).

Example 9-7. Linker Allocation Without HIGH Specifier

```
.bss       0    00000200    00000270     UNINITIALIZED
                  00000200    0000011a     rtsxxx.lib  : defs.obj (.bss)
                  0000031a    00000088                 : trgdrv.obj (.bss)
                  000003a2    00000078                 : lowlev.obj (.bss)
                  0000041a    00000046                 : exit.obj (.bss)
                  00000460    00000008                 : memory.obj (.bss)
                  00000468    00000004                 : _lock.obj (.bss)
                  0000046c    00000002                 : fopen.obj (.bss)
                  0000046e    00000002     hello.obj (.bss)

.stack     0    00000470    00000140     UNINITIALIZED
                  00000470    00000002     rtsxxx.lib  : boot.obj (.stack)

.sysmem    0    000005b0    00000120     UNINITIALIZED
                  000005b0    00000004     rtsxxx.lib  : memory.obj (.sysmem)
```

9.6.4.2.4 Alignment and Blocking

You can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the align keyword. For example, the following code allocates .text so that it falls on a 32-byte boundary:

```
.text: load = align(32)
```

You can specify the same alignment with the palign keyword. In addition, palign ensures the section's size is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

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

```
bss: load = block(0x0080)
```

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

9.6.4.2.5 Alignment With Padding

As with align, you can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the palign keyword. In addition, palign ensures that the size of the section is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

For example, the following code lines allocate .text on a 2-byte boundary within the PMEM area. The .text section size is guaranteed to be a multiple of 2 bytes. Both statements are equivalent:

```
.text: palign(2) {} > PMEM

.text: palign = 2 {} > PMEM
```

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

For example, consider the following section specification:

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

In this example, the length of the .mytext section is 6 bytes before the palign operator is applied. The contents of .mytext are as follows:

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
```

After the palign operator is applied, the length of .mytext is 8 bytes, and its contents are as follows:

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0xffff
```

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

The fill value specified in the linker command file is interpreted as a 16-bit constant, so if you specify this code:

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

The fill value assumed by the linker is 0x00ff, and .mytext will then have the following contents:

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0xffff
0008 0x00ff
000a 0x00ff
```

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

The palign operator can also take a parameter of *power2*. This parameter tells the linker to add padding to increase the section's size to the next power of two boundary. In addition, the section is aligned on that power of 2 as well.

For example, consider the following section specification:

```
    .mytext: palign(power2) {} > PMEM
```

Assume that the size of the .mytext section is 120 bytes and PMEM starts at address 0x10020. After applying the palign(power2) operator, the .mytext output section will have the following properties:

```
  name     addr          size     align
-------    ----------    -----    -----
.mytext    0x00010080    0x80     128
```

9.6.4.2.6 Using the Page Method

Using the page method of specifying an address, you can allocate a section into an address space that is named in the MEMORY directive. For example:

```
MEMORY
{
   PAGE 0 :  PROG    : origin = 0x00000800,   length = 0x00240
   PAGE 1 :  DATA    : origin = 0x00000A00,   length = 0x02200
   PAGE 1 :  OVR_MEM : origin = 0x00002D00,   length = 0x01000
   PAGE 2 :  DATA    : origin = 0x00000A00,   length = 0x02200
   PAGE 2 :  OVR_MEM : origin = 0x00002D00,   length = 0x01000
}
SECTIONS
{
   .text:   PAGE = 0
   .data:   PAGE = 2
   .cinit:  PAGE = 0
   .bss:    PAGE = 1
}
```

In this example, the .text and .cinit sections are allocated to PAGE 0. They are placed anywhere within the bounds of PAGE 0. The .data section is allocated anywhere within the bounds of PAGE 2. The .bss section is allocated anywhere within the bounds of PAGE 1.

You can use the page method in conjunction with any of the other methods to restrict an allocation to a specific address space. For example:

```
   .text: load = OVR_MEM PAGE 1
```

In this example, the .text section is allocated to the named memory range OVR_MEM. There are two named memory ranges called OVR_MEM, however, so you must specify which one is to be used. By adding PAGE 1, you specify the use of the OVR_MEM memory range in address space PAGE 1 rather than in address space PAGE 2. If no PAGE is specified for a section, the linker allocates initialized sections to PAGE 0 and uninitialized sections to PAGE 1.

9.6.4.3 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. In general, the linker combines input sections by concatenating them in the order in which they are specified. However, if alignment or blocking is specified for an input section, all of the input sections within the output section are ordered as follows:

- All aligned sections, from largest to smallest
- All blocked sections, from largest to smallest
- All other sections, from largest to smallest

The size of an output section is the sum of the sizes of the input sections that it comprises.

Example 9-8 shows the most common type of section specification; note that no input sections are listed.

Example 9-8. The Most Common Method of Specifying Section Contents

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

In Example 9-8, 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 9-8 are actually a shorthand method for the following:

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

The specification *(.text) means *the unallocated .text sections from all the input files.* This format is useful when:

- You want the output section to contain all input sections that have a specified name, but the output section name is different from the input sections' name.
- You want the linker to allocate the input sections before it processes additional input sections or commands within the braces.

The following example illustrates the 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.

9.6.4.4 Using Multi-Level Subsections

Subsections can be identified with the base section name and one or more subsection names separated by colons. For example, A:B and A:B:C name subsections of the base section A. In certain places in a link command file specifying a base name, such as A, selects the section A as well as any subsections of A, such as A:B or A:C:D.

A name such as A:B can be used to specify a (sub)section of that name as well as any (multi-level) subsections beginning with that name, such as A:B:C, A:B:OTHER, etc. All the subsections of A:B are also subsections of A. A and A:B are supersections of A:B:C. Among a group of supersections of a subsection, the nearest supersection is the supersection with the longest name. Thus, among {A, A:B} the nearest supersection of A:B:C:D is A:B.

With multiple levels of subsections, the constraints are the following:

1. When specifying **input** sections within a file (or library unit) the section name selects an input section of the same name and any subsections of that name.
2. Input sections that are not explicitly allocated are allocated in an existing **output** section of the same name or in the nearest existing supersection of such an output section. An exception to this rule is that during a partial link (specified by the --relocatable linker option) a subsection is allocated only to an existing output section of the same name.
3. If no such output section described in 2) is defined, the input section is put in a **newly created output** section with the same name as the base name of the input section

Consider linking input sections with the following names:

europe:north:norway	europe:central:france	europe:south:spain
europe:north:sweden	europe:central:germany	europe:south:italy
europe:north:finland	europe:central:denmark	europe:south:malta
europe:north:iceland		

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS {
  nordic:  {*(europe:north)
            *(europe:central:denmark)} /* the nordic countries */
  central: {*(europe:central)}         /* france, germany      */
  therest: {*(europe)}                 /* spain, italy, malta  */
}
```

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS {
  islands: {*(europe:south:malta)
            *(europe:north:iceland)}  /* malta, iceland   */
  europe:north:finland : {}           /* finland          */
  europe:north         : {}           /* norway, sweden   */
  europe:central       : {}           /* germany, denmark */
  europe:central:france: {}           /* france           */

  /* (italy, spain) go into a linker-generated output section "europe" */
}
```

Upward Compatibility of Multi-Level Subsections

NOTE: Existing linker commands that use the existing single-level subsection features and which do not contain section names containing multiple colon characters continue to behave as before. However, if section names in a link command file or in the input sections supplied to the linker contain multiple colon characters, some change in behavior could be possible. You should carefully consider the impact of the new rules for multiple levels to see if it affects a particular system link.

9.6.4.5 Specifying Library or Archive Members as Input to Output Sections

You can specify one or more members of an object library or archive for input to an output section. Consider this SECTIONS directive:

Example 9-9. Archive Members to Output Sections

```
SECTIONS
{
        boot    >        BOOT1
        {
                -l=rtsXX.lib<boot.obj> (.text)
                -l=rtsXX.lib<exit.obj strcpy.obj> (.text)
        }

        .rts    >        BOOT2
        {
                -l=rtsXX.lib (.text)
        }

        .text    >        RAM
        {
                * (.text)
        }
}
```

In Example 9-9, the .text sections of boot.obj, exit.obj, and strcpy.obj are extracted from the run-time-support library and placed in the .boot output section. The remainder of the run-time-support library object that is referenced is allocated to the .rts output section. Finally, the remainder of all other .text sections are to be placed in section .text.

An archive member or a list of members is specified by surrounding the member name(s) with angle brackets < and > after the library name. Any object files separated by commas or spaces from the specified archive file are legal within the angle brackets.

The --library option (which normally implies a library path search be made for the named file following the option) listed before each library in Example 9-9 is optional when listing specific archive members inside < >. Using < > implies that you are referring to a library.

To collect a set of the input sections from a library in one place, use the --library option within the SECTIONS directive. For example, the following collects all the .text sections from rts55.lib into the .rtstest section:

```
SECTIONS
{
   .rtstest { -l=rts55.lib(.text) } > RAM
}
```

SECTIONS Directive Effect on --priority

NOTE: Specifying a library in a SECTIONS directive causes that library to be entered in the list of libraries that the linker searches to resolve references. If you use the --priority option, the first library specified in the command file will be searched first.

9.6.4.6 Allocation Using Multiple Memory Ranges

The linker allows you to specify an explicit list of memory ranges into which an output section can be allocated. Consider the following example:

```
MEMORY
{
   P_MEM1 :  origin = 0x02000,  length = 0x01000
   P_MEM2 :  origin = 0x04000,  length = 0x01000
   P_MEM3 :  origin = 0x06000,  length = 0x01000
   P_MEM4 :  origin = 0x08000,  length = 0x01000
}

SECTIONS
{
   .text  : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

The | operator is used to specify the multiple memory ranges. The .text output section is allocated as a whole into the first memory range in which it fits. The memory ranges are accessed in the order specified. In this example, the linker first tries to allocate the section in P_MEM1. If that attempt fails, the linker tries to place the section into P_MEM2, and so on. If the output section is not successfully allocated in any of the named memory ranges, the linker issues an error message.

With this type of SECTIONS directive specification, the linker can seamlessly handle an output section that grows beyond the available space of the memory range in which it is originally allocated. Instead of modifying the link command file, you can let the linker move the section into one of the other areas.

9.6.4.7 Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges

The linker can split output sections among multiple memory ranges to achieve an efficient allocation. Use the >> operator to indicate that an output section can be split, if necessary, into the specified memory ranges. For example:

```
MEMORY
{
   P_MEM1 :  origin = 0x2000,  length = 0x1000
   P_MEM2 :  origin = 0x4000,  length = 0x1000
   P_MEM3 :  origin = 0x6000,  length = 0x1000
   P_MEM4 :  origin = 0x8000,  length = 0x1000
}

SECTIONS
{
  .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

In this example, the >> operator indicates that the .text output section can be split among any of the listed memory areas. If the .text section grows beyond the available memory in P_MEM1, it is split on an input section boundary, and the remainder of the output section is allocated to P_MEM2 | P_MEM3 | P_MEM4.

The | operator is used to specify the list of multiple memory ranges.

You can also use the >> operator to indicate that an output section can be split within a single memory range. This functionality is useful when several output sections must be allocated into the same memory range, but the restrictions of one output section cause the memory range to be partitioned. Consider the following example:

```
MEMORY
{
   RAM :  origin = 0x1000,  length = 0x8000
}

SECTIONS
{
  .special: { f1.obj(.text) } load = 0x4000
  .text: { *(.text) } >> RAM
}
```

The .special output section is allocated near the middle of the RAM memory range. This leaves two unused areas in RAM: from 0x1000 to 0x4000, and from the end of f1.obj(.text) to 0x8000. The specification for the .text section allows the linker to split the .text section around the .special section and use the available space in RAM on either side of .special.

The >> operator can also be used to split an output section among all memory ranges that match a specified attribute combination. For example:

```
MEMORY
{
  P_MEM1 (RWX) : origin = 0x1000,  length = 0x2000
  P_MEM2 (RWI) : origin = 0x4000,  length = 0x1000
}

SECTIONS
{
  .text: { *(.text) } >> (RW)
}
```

The linker attempts to allocate all or part of the output section into any memory range whose attributes match the attributes specified in the SECTIONS directive.

This SECTIONS directive has the same effect as:

```
SECTIONS
{
.text: { *(.text) } >> P_MEM1 | P_MEM2}
}
```

Certain sections should not be split:

- Certain sections created by the compiler, including
  - The .cinit section, which contains the autoinitialization table for C/C++ programs
  - The .pinit section, which contains the list of global constructors for C++ programs
  - The .bss section, which defines global variables
  - The .sysmem, .stack, and .sysstack sections, which are uninitialized sections for the C memory pool used by the malloc() functions and the run-time stacks, respectively.
- An output section with separate load and run allocations. The code that copies the output section from its load-time allocation to its run-time location cannot accommodate a split in the output section.
- An output section with an input section specification that includes an expression to be evaluated. The expression may define a symbol that is used in the program to manage the output section at run time.
- An output section that has a START(), END(), OR SIZE() operator applied to it. These operators provide information about a section's load or run address, and size. Splitting the section may compromise the integrity of the operation.
- The run allocation of a UNION. (Splitting the load allocation of a UNION is allowed.)

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

9.6.5 Specifying a Section's Run-Time Address

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

The linker provides a simple way to accomplish this. You can use the SECTIONS directive to direct the linker to allocate a section twice: once to set its load address and again to set its run address. For example:

```
.fir: load = SLOW_MEM, run = FAST_MEM
```

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

See Section 2.6 for an overview on run-time relocation.

9.6.5.1 Specifying Load and Run Addresses

The load address determines where a loader places the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this does *not* happen automatically when you specify a separate run address. (The TABLE operator instructs the linker to produce a copy table; see Section 9.9.5.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is allocated as if it were two sections of the same size. This means that both allocations occupy space in the memory map and cannot overlay each other or other sections. (The UNION directive provides a way to overlay sections; see Section 9.6.6.1.)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. Everything related to allocation after the keyword *load* affects the load address until the keyword *run* is seen, after which, everything affects the run address. The load and run allocations are completely independent, so any qualification of one (such as alignment) has no effect on the other. You can also specify run first, then load. Use parentheses to improve readability.

The examples below specify load and run addresses:

```
.data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

(align applies only to load)

```
.data: load = (SLOW_MEM align 32), run = FAST_MEM
```

(identical to previous example)

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

(align 32 in FAST_MEM for run; align 16 anywhere for load)

For more information on run-time relocation see

9.6.5.2 Uninitialized Sections

Uninitialized sections (such as .bss) are not loaded, so their only significant address is the run address. The linker allocates uninitialized sections only once: if you specify both run and load addresses, the linker warns you and ignores the load address. Otherwise, if you specify only one address, the linker treats it as a run address, regardless of whether you call it load or run. This example specifies load and run addresses for an uninitialized section:

```
.bss: load = 0x1000, run = FAST_MEM
```

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

```
.bss: load = FAST_MEM
.bss: run = FAST_MEM
.bss: > FAST_MEM
```

9.6.5.3 Referring to the Load Address by Using the .label Directive

Normally, any reference to a symbol in a section refers to its run-time address. However, it may be necessary at run time to refer to a load-time address. Specifically, the code that copies a section from its load address to its run address must have access to the load address. The .label directive defines a special symbol that refers to the section's load address. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address. See Create a Load-Time Address Label for more information on the .label directive.

Example 9-10 and Example 9-11 show the use of the .label directive to copy a section from its load address in ROM to its run address in RAM. Figure 9-4 illustrates the run-time execution of Example 9-10.

The table operator and cpy_in can also be used to refer to a load address; see Section 9.9.5.

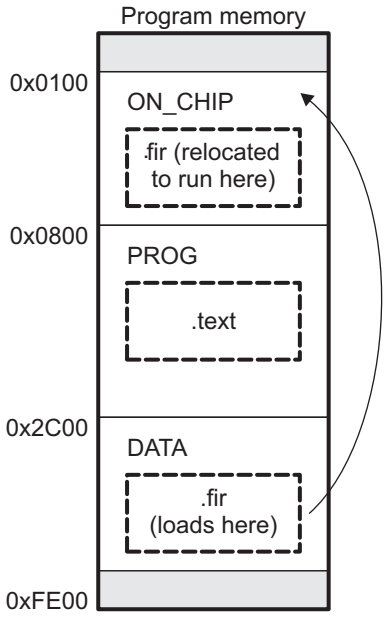
Example 9-10. Copying Section Assembly Language File

```
;  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

     MOV #fir_src,AR1        ; get load address
     MOV BRC0,T1
     MOV T1,BRC1
     MOV #(fir_end - fir_src - 1),BRC0
     RPTB end
end  MOV *AR1+,*CDP+
     MOV BRC1,T1
     MOV T1,BRC0

;  jump to section, now in RAM
     CALL fir
```

Example 9-11. Linker Command File for Example 9-10

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

MEMORY
{
   ONCHIP :  origin = 0x0100,  length = 0x0700
   PROG   :  origin = 0x0800,  length = 0x2400
   DATA   :  origin = 0x2C00,  length = 0xD200
}

SECTIONS
{
   .text: load = PROG
   .fir:  load = DATA, run ONCHIP
}
```

Figure 9-4. Run-Time Execution of Example 9-10



9.6.6 Using UNION and GROUP Statements

Two SECTIONS statements allow you to conserve memory: GROUP and UNION. Unioning sections causes the linker to allocate them to the same run address. Grouping sections causes the linker to allocate them contiguously in memory. Section names can refer to sections, subsections, or archive library members.

9.6.6.1 Overlaying Sections With the UNION Statement

For some applications, you may want to allocate more than one section to occupy the same address during run time. For example, you may have several routines you want in fast external memory at various stages of execution. Or you may want several data objects that are not active at the same time to share a block of memory. The UNION statement within the SECTIONS directive provides a way to allocate several sections at the same run-time address.

In Example 9-12, the .bss sections from file1.obj and file2.obj are allocated at the same address in FAST_MEM. In the memory map, the union occupies as much space as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

Example 9-12. The UNION Statement

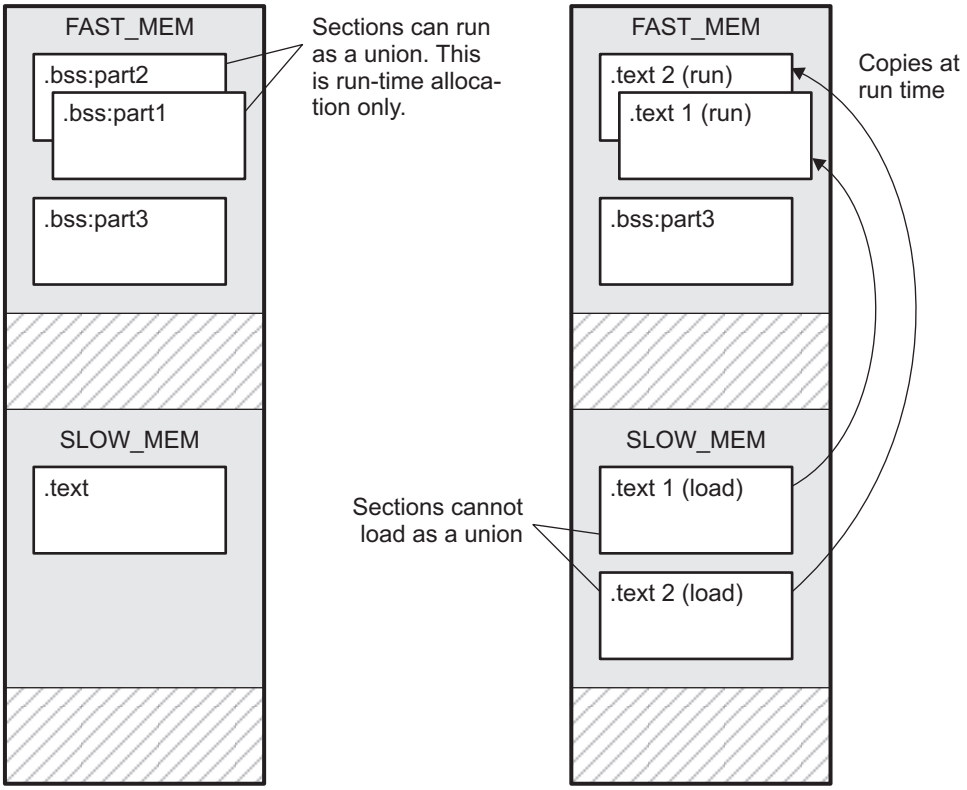
```
SECTIONS
{
   .text: load = SLOW_MEM
   UNION: run  = FAST_MEM
   {
      .bss:part1: { file1.obj(.bss) }
      .bss:part2: { file2.obj(.bss) }
   }
      .bss:part3: run = FAST_MEM { globals.obj(.bss) }
}
```

Allocation of a section as part of a union affects only its *run address.* Under no circumstances can sections be overlaid for loading. If an initialized section is a union member (an initialized section, such as .text, has raw data), its load allocation *must* be separately specified. See Example 9-13.

Example 9-13. Separate Load Addresses for UNION Sections

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

Figure 9-5. Memory Allocation Shown in Example 9-12 and Example 9-13



Since the .text sections contain raw data, they cannot *load* as a union, although they can be *run* as a union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a UNION, the linker issues a warning and allocates load space anywhere it can in configured memory.

Uninitialized sections are not loaded and do not require load addresses.

The UNION statement applies only to allocation of run addresses, so it is meaningless to specify a load address for the union itself. For purposes of allocation, the union is treated as an uninitialized section: any one allocation specified is considered a run address, and if both run and load addresses are specified, the linker issues a warning and ignores the load address.

UNION and Overlay Page Are Not the Same

NOTE: The UNION capability and the overlay page capability (see Section 9.6.7) 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.

9.6.6.2 Grouping Output Sections Together

The SECTIONS directive's GROUP option forces several output sections to be allocated contiguously. For example, assume that a section named term_rec contains a termination record for a table in the .data section. You can force the linker to allocate .data and term_rec together:

Example 9-14. Allocate Sections Together

```
SECTIONS
{
   .text           /* Normal output section             */
   .bss            /* Normal output section             */
   GROUP 0x00001000 : /* Specify a group of sections    */
   {
      .data        /* First section in the group        */
      term_rec     /* Allocated immediately after .data */
   }
}
```

You can use binding, alignment, or named memory to allocate a GROUP in the same manner as a single output section. In the preceding example, the GROUP is bound to address 0x1000. This means that .data is allocated at 0x1000, and term_rec follows it in memory.

You Cannot Specify Addresses for Sections Within a GROUP

NOTE: 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.

9.6.6.3 Nesting UNIONs and GROUPs

The linker allows arbitrary nesting of GROUP and UNION statements with the SECTIONS directive. By nesting GROUP and UNION statements, you can express hierarchical overlays and groupings of sections. Example 9-15 shows how two overlays can be grouped together.

Example 9-15. Nesting GROUP and UNION Statements

```
SECTIONS
{
   GROUP 0x1000 : run = FAST_MEM
   {
      UNION:
      {
         mysect1: load = SLOW_MEM
         mysect2: load = SLOW_MEM
      }
      UNION:
      {
         mysect3: load = SLOW_MEM
         mysect4: load = SLOW_MEM
      }
   }
}
```

For this example, the linker performs the following allocations:

- The four sections (mysect1, mysect2, mysect3, mysect4) are assigned unique, non-overlapping load addresses. The name you defined with the .label directive is used in the SLOW_MEM memory region. This assignment is determined by the particular load allocations given for each section.
- Sections mysect1 and mysect2 are assigned the same run address in FAST_MEM.
- Sections mysect3 and mysect4 are assigned the same run address in FAST_MEM.
- The run addresses of mysect1/mysect2 and mysect3/mysect4 are allocated contiguously, as directed by the GROUP statement (subject to alignment and blocking restrictions).

To refer to groups and unions, linker diagnostic messages use the notation:

GROUP_*n* UNION_*n*

In this notation, *n* is a sequential number (beginning at 1) that represents the lexical ordering of the group or union in the linker control file, without regard to nesting. Groups and unions each have their own counter.

9.6.6.4 Checking the Consistency of Allocators

The linker checks the consistency of load and run allocations specified for unions, groups, and sections. The following rules are used:

- Run allocations are only allowed for top-level sections, groups, or unions (sections, groups, or unions that are not nested under any other groups or unions). The linker uses the run address of the top-level structure to compute the run addresses of the components within groups and unions.
- The linker does not accept a load allocation for UNIONs.
- The linker does not accept a load allocation for uninitialized sections.
- In most cases, you must provide a load allocation for an initialized section. However, the linker does not accept a load allocation for an initialized section that is located within a group that already defines a load allocator.
- As a shortcut, you can specify a load allocation for an entire group, to determine the load allocations for every initialized section or subgroup nested within the group. However, a load allocation is accepted for an entire group only if all of the following conditions are true:
  - The group is initialized (that is, it has at least one initialized member).
  - The group is not nested inside another group that has a load allocator.
  - The group does not contain a union containing initialized sections.
- If the group contains a union with initialized sections, it is necessary to specify the load allocation for each initialized section nested within the group. Consider the following example:

```
SECTIONS
{
  GROUP: load = SLOW_MEM, run = SLOW_MEM
  {
   .text1:
   UNION:
   {
    .text2:
    .text3:
   }
  }
}
```

The load allocator given for the group does not uniquely specify the load allocation for the elements within the union: .text2 and .text3. In this case, the linker issues a diagnostic message to request that these load allocations be specified explicitly.

9.6.6.5 Naming UNIONs and GROUPs

You can give a name to a UNION or GROUP by entering the name in parentheses after the declaration. For example:

```
GROUP(BSS_SYSMEM_STACK_GROUP)
{
   .bss    :{}
   .sysmem :{}
   .stack  :{}
} load=D_MEM,  run=D_MEM
```

The name you defined is used in diagnostics for easy identification of the problem LCF area. For example:

```
warning: LOAD placement ignored for "BSS_SYSMEM_STACK_GROUP": object is uninitialized

   UNION(TEXT_CINIT_UNION)
   {
      .const :{}load=D_MEM, table(table1)
      .pinit :{}load=D_MEM, table(table1)
   }run=P_MEM

warning:table(table1) operator ignored: table(table1) has already been applied to a section
in the "UNION(TEXT_CINIT_UNION)" in which ".pinit" is a descendant
```

9.6.7 Overlaying Pages

Some devices 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 then use the SECTIONS directive to specify the sections to be mapped into various pages.

Overlay Section and Overlay Page Are Not the Same

NOTE: The UNION capability and the overlay page capability (see Section 9.6.6.1) 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 it is cumbersome.

9.6.7.1 Using the MEMORY Directive to Define Overlay Pages

To the linker, each overlay page represents a completely separate memory space comprising the full range of addressable locations. In this way, you can 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).

9.6.7.2 Example of Overlay Pages

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 2BFFh for PAGE 2. Although only one bank can be selected at a time, you can initialize each bank with different data. Example 9-16 shows how you use the MEMORY directive to obtain this configuration:

Example 9-16. MEMORY Directive With Overlay Pages

```
MEMORY
{
   PAGE 0  : RAM      :origin = 0x0800, length = 0x0240
             : PROG     :origin = 0x2C00, length = 0xD200
   PAGE 1  : OVR_MEM  :origin = 0x0A00, length = 0x2200
             : DATA     :origin = 0x2C00, length = 0xD400
   PAGE 2  : OVR_MEM  :origin = 0x0A00, length = 0x2200
}
```

Example 9-16 defines three separate address spaces.

- PAGE 0 defines an area of RAM program memory space 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.

9.6.7.3 Using Overlay Pages With the SECTIONS Directive

Assume that you are using the MEMORY directive as shown in Example 9-16. Further assume that your code consists of the standard sections, as well as four modules of code that you want to load in data memory space and run in RAM program memory. Example 9-17 shows how to use the SECTIONS directive overlays to accomplish these objectives.

Example 9-17. SECTIONS Directive Definition for Overlays in Example 7-10

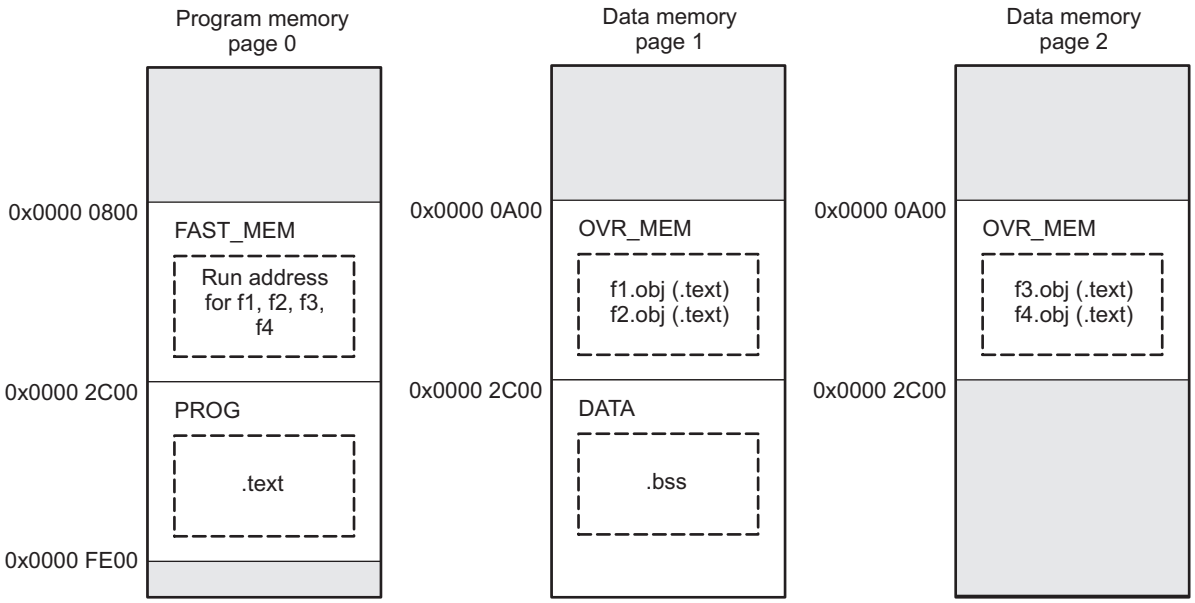
```
SECTIONS
{
   UNION :  run = RAM
   {
      S1 :  load = OVR_MEM PAGE 1
      {
         s1_load = 0x0A00h;
         s1_start = .;
         f1.obj (.text)
         f2.obj (.text)
         s1_length = . - s1_start;
      }
      S2 :  load = OVR_MEM PAGE 2
      {
         s2_load = 0x0A00h;
         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 are f1, f2, f3, and f4. 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.

9.6.7.4 Memory Allocation for Overlaid Pages

Figure 9-6 shows overlay pages defined by the MEMORY directive in Example 9-16 and the SECTIONS directive in Example 9-17.

Figure 9-6. Overlay Pages Defined in Example 9-16 and Example 9-17



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

You can assign three special types to output sections: DSECT, COPY, and NOLOAD. These types affect the way that the program is treated when it is linked and loaded. You can assign a type to a section by placing the type after the section definition. For example:

```
SECTIONS
{
   sec1: load = 0x00002000, type =  DSECT  {f1.obj}
   sec2: load = 0x00004000, type =  COPY   {f2.obj}
   sec3: load = 0x00006000, type =  NOLOAD {f3.obj}
}
```

- The DSECT type creates a dummy section with the following characteristics:
  - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.
  - It can overlay other output sections, other DSECTs, and unconfigured memory.
  - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.
  - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.
  - The section's contents, relocation information, and line number information are not placed in the output module.

  In the preceding example, none of the sections from f1.obj are allocated, but all the symbols are relocated as though the sections were linked at address 0x2000. The other sections can refer to any of the global symbols in sec1.

- A COPY section is similar to a DSECT section, except that its contents and associated information are written to the output module. The .cinit section that contains initialization tables for the TMS320C55x C/C++ compiler has this attribute under the run-time initialization model.
- A NOLOAD section differs from a normal output section in one respect: the section's contents, relocation information, and line number information are not placed in the output module. The linker allocates space for the section, and it appears in the memory map listing.

9.6.9 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.

9.6.9.1 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 Section 9.6.9.3. Assignment statements *must* terminate with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Therefore, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol cur_tab as the address of the current table. The cur_tab symbol must point to either Table1 or Table2. You could accomplish this in the assembly code, but you would need to reassemble the program to change tables. Instead, you can use a linker assignment statement to assign cur_tab at link time:

```
prog.obj           /* Input file                          */
cur_tab = Table1;  /* Assign cur_tab to one of the tables */
```

9.6.9.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the section program counter (SPC) during allocation. The SPC keeps track of the current location within a section. The linker's . symbol is analogous to the assembler's $ symbol. The . symbol can be used only in assignment statements within a SECTIONS directive because . is meaningful only during allocation and SECTIONS controls the allocation process. (See Section 9.6.4.)

The . symbol refers to the current run address, not the current load address, of the section.

For example, suppose a program needs to know the address of the beginning of the .data section. By using the .global directive (see Identify Global Symbols), you can create an external undefined variable called Dstart in the program. Then, assign the value of . to Dstart:

```
SECTIONS
{
   .text:    {}
   .data:    {Dstart = .;}
   .bss :    {}
}
```

This defines Dstart to be the first linked address of the .data section. (Dstart is assigned *before* .data is allocated.) The linker relocates all references to Dstart.

A special type of assignment assigns a value to the . symbol. This adjusts the SPC within an output section and creates a hole between two input sections. Any value assigned to . to create a hole is relative to the beginning of the section, not to the address actually represented by the . symbol. Holes and assignments to . are described in Section 9.6.10.

9.6.9.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in .
- All numbers are treated as long (32-bit) integers.
- Constants are identified by the linker in the same way as by the assembler. That is, numbers are recognized as decimal unless they have a suffix (H or h for hexadecimal and Q or q for octal). C language prefixes are also recognized (0 for octal and 0x for hex). Hexadecimal constants must begin with a digit. No binary constants are allowed.
- Symbols within an expression have only the value of the symbol's *address*. No type-checking is performed.
- Linker expressions can be absolute or relocatable. If an expression contains *any* relocatable symbols (and 0 or more constants or absolute symbols), it is relocatable. Otherwise, the expression is absolute. If a symbol is assigned the value of a relocatable expression, it is relocatable; if it is assigned the value of an absolute expression, it is absolute.

The linker supports the C language operators listed in Table 9-9 in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in , the linker also has an align operator that allows a symbol to be aligned on an n-byte boundary within an output section (n is a power of 2). For example, the following expression aligns the SPC within the current section on the next 16-byte boundary. Because the align operator is a function of the current SPC, it can be used only in the same context as . —that is, within a SECTIONS directive.

```
. = align(16);
```

Table 9-9. Operators Used in Expressions

Symbols	Operators	Evaluation
+ - ~ (1)	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

(1) Unary +, -, and * have higher precedence than the binary forms.

9.6.9.4 Symbols Defined by the Linker

The linker automatically defines several symbols based on which sections are used in your assembly source. A program can use these symbols at run time to determine where a section is linked. Since these symbols are external, they appear in the linker map. Each symbol can be accessed in any assembly language module if it is declared with a .global directive (see Identify Global Symbols). You must have used the corresponding section in a source module for the symbol to be created. Values are assigned to these symbols as follows:

.text	is assigned the first address of the .text output section. (It marks the *beginning* of executable code.)
etext	is assigned the first address following the .text output section. (It marks the *end* of executable code.)
.data	is assigned the first address of the .data output section. (It marks the *beginning* of initialized data tables.)
edata	is assigned the first address following the .data output section. (It marks the *end* of initialized data tables.)
.bss	is assigned the first address of the .bss output section. (It marks the *beginning* of uninitialized data.)
end	is assigned the first address following the .bss output section. (It marks the *end* of uninitialized data.)

The following symbols are defined only for C/C++ support when the --ram_model or --rom_model option is used.

__STACK_END	is assigned the end of the .stack size.
__STACK_SIZE	is assigned the size of the .stack section.
_SYSSTACK_SIZE	is assigned the size of the .sysstack section.
__SYSMEM_SIZE	is assigned the size of the .sysmem section.

Allocation of .stack and .sysstack Sections

NOTE: The .stack and .sysstack sections must be allocated on the same 64K-word data page.

9.6.9.5 Assigning Exact Start, End, and Size Values of a Section to a Symbol

The code generation tools currently support the ability to load program code in one area of (slow) memory and run it in another (faster) area. This is done by specifying separate load and run addresses for an output section or group in the link command file. Then execute a sequence of instructions (the copying code in Example 9-10) that moves the program code from its load area to its run area before it is needed.

There are several responsibilities that a programmer must take on when setting up a system with this feature. One of these responsibilities is to determine the size and run-time address of the program code to be moved. The current mechanisms to do this involve use of the .label directives in the copying code. A simple example is illustrated Example 9-10.

This method of specifying the size and load address of the program code has limitations. While it works fine for an individual input section that is contained entirely within one source file, this method becomes more complicated if the program code is spread over several source files or if the programmer wants to copy an entire output section from load space to run space.

Another problem with this method is that it does not account for the possibility that the section being moved may have an associated far call trampoline section that needs to be moved with it.

9.6.9.6 Why the Dot Operator Does Not Always Work

The dot operator (.) is used to define symbols at link-time with a particular address inside of an output section. It is interpreted like a PC. Whatever the current offset within the current section is, that is the value associated with the dot. Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.obj(.text)
        end_of_s2 = .;
}
```

This statement creates three symbols:

- end_of_s1—the end address of .text in s1.obj
- start_of_s2—the start address of .text in s2.obj
- end_of_s2—the end address of .text in s2.obj

Suppose there is padding between s1.obj and s2.obj that is created as a result of alignment. Then start_of_s2 is not really the start address of the .text section in s2.obj, but it is the address before the padding needed to align the .text section in s2.obj. This is due to the linker's interpretation of the dot operator as the current PC. It is also due to the fact that the dot operator is evaluated independently of the input sections around it.

Another potential problem in the above example is that end_of_s2 may not account for any padding that was required at the end of the output section. You cannot reliably use end_of_s2 as the end address of the output section. One way to get around this problem is to create a dummy section immediately after the output section in question. For example:

```
GROUP
{
   outsect:
   {
      start_of_outsect = .;
      ...
   }
      dummy: { size_of_outsect = . - start_of_outsect; }
}
```

9.6.9.7 Address and Dimension Operators

Six new operators have been added to the link command file syntax:

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

Linker Command File Operator Equivalencies

NOTE: LOAD_START() and START() are equivalent, as are LOAD_END()/END() and LOAD_SIZE()/SIZE(). The LOAD names are recommended for clarity.

The new address and dimension operators can be associated with several different kinds of allocation units, including input items, output sections, GROUPs, and UNIONs. The following sections provide some examples of how the operators can be used in each case.

9.6.9.7.1 Input Items

Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.obj(.text)
        end_of_s2 = .;
}
```

This can be rewritten using the START and END operators as follows:

```
outsect:
{
        s1.obj(.text) { END(end_of_s1) }
        s2.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

The values of end_of_s1 and end_of_s2 will be the same as if you had used the dot operator in the original example, but start_of_s2 would be defined after any necessary padding that needs to be added between the two .text sections. Remember that the dot operator would cause start_of_s2 to be defined before any necessary padding is inserted between the two input sections.

The syntax for using these operators in association with input sections calls for braces { } to enclose the operator list. The operators in the list are applied to the input item that occurs immediately before the list.

9.6.9.7.2 Output Section

The START, END, and SIZE operators can also be associated with an output section. Here is an example:

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
        <list of input items>
}
```

In this case, the SIZE operator defines size_of_outsect to incorporate any padding that is required in the output section to conform to any alignment requirements that are imposed.

The syntax for specifying the operators with an output section does not require braces to enclose the operator list. The operator list is simply included as part of the allocation specification for an output section.

9.6.9.7.3 GROUPs

Here is another use of the START and SIZE operators in the context of a GROUP specification:

```
GROUP
{
        outsect1: { ... }
        outsect2: { ... }
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

This can be useful if the whole GROUP is to be loaded in one location and run in another. The copying code can use group_start and group_size as parameters for where to copy from and how much is to be copied. This makes the use of .label in the source code unnecessary.

9.6.9.7.4 UNIONs

The RUN_SIZE and LOAD_SIZE operators provide a mechanism to distinguish between the size of a UNION's load space and the size of the space where its constituents are going to be copied before they are run. Here is an example:

```
UNION: run = RAM, LOAD_START(union_load_addr),
       LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
        .text1: load = ROM, SIZE(text1_size) { f1.obj(.text) }
        .text2: load = ROM, SIZE(text2_size) { f2.obj(.text) }
}
```

Here union_ld_sz is going to be equal to the sum of the sizes of all output sections placed in the union. The union_run_sz value is equivalent to the largest output section in the union. Both of these symbols incorporate any padding due to blocking or alignment requirements.

9.6.10 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called *holes*. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles holes and how you can fill holes (and uninitialized sections) with values.

9.6.10.1 Initialized and Uninitialized Sections

There are two rules to remember about the contents of output sections. An output section contains either:

- Raw data for the *entire* section
- *No* raw data

A section that has raw data is referred to as *initialized*. This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections *always* have raw data if anything was assembled into them. Named sections defined with the .sect assembler directive also have raw data.

By default, the .bss section (see Reserve Space in the .bss Section) and sections defined with the .usect directive (see Reserve Uninitialized Space) have no raw data (they are *uninitialized*). They occupy space in the memory map but have no actual contents. Uninitialized sections typically reserve space in fast external memory for variables. In the object file, an uninitialized section has a normal section header and can have symbols defined in it; no memory image, however, is stored in the section.

9.6.10.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must supply raw data for the hole.*

Holes can be created only *within* output sections. Space can exist *between* output sections, but such space is not a hole. To fill the space between output sections, see Section 9.6.3.2.

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 9.6.9.

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

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
     . += 0x0100    /* Create a hole with size 0x0100 */
      file2.obj(.text)
    . = align(16);  /* Create a hole to align the SPC */
      file3.obj(.text)
    }
}
```

The output section outsect is built as follows:

1. The .text section from file1.obj is linked in.
2. The linker creates a 256-byte hole.
3. The .text section from file2.obj is linked in after the hole.
4. The linker creates another hole by aligning the SPC on a 16-byte boundary.
5. Finally, the .text section from file3.obj is linked in.

All values assigned to the . symbol within a section refer to the *relative address within the section.* The linker handles assignments to the . symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement . = align(16) in the example. This statement effectively aligns the file3.obj .text section to start on a 16-byte boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, the file3.obj .text section will not be aligned either.

The . symbol refers to the current run address, not the current load address, of the section.

Expressions that decrement the . symbol are illegal. For example, it is invalid to use the -= operator in an assignment to the . symbol. The most common operators used in assignments to the . symbol are += and align.

If an output section contains all input sections of a certain type (such as .text), you can use the following statements to create a hole at the beginning or end of the output section.

```
    .text:   {  .+= 0x0100; }     /* Hole at the beginning */
    .data:   {  *(.data)
                . += 0x0100; }    /* Hole at the end       */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* The following example illustrates this method:

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
     file1.obj(.bss)           /* This becomes a hole */
   }
}
```

Because the .text section has raw data, all of outsect must also contain raw data. Therefore, the uninitialized .bss section becomes a hole.

Uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

9.6.10.3 Filling Holes

When a hole exists in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 32-bit fill value that is replicated through memory until it fills the hole. The linker determines the fill value as follows:

1. If the hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = sign and a 32-bit constant. For example:

```
SECTIONS
{  outsect:
   {
      file1.obj(.text)
      file2.obj(.bss)= 0xFF00 /* Fill this hole with 0xFF00 */
   }
}
```

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 = 0xFF00         /* Fills holes with 0xFF00     */
   {
      . += 0x0010;                 /* This creates a hole         */
      file1.obj(.text)
      file1.obj(.bss)              /* This creates another hole   */
   }
}
```

3. If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with the --fill_value option (see Section 9.4.9). For example, suppose the command file link.cmd contains the following SECTIONS directive:

```
SECTIONS { .text: { .= 0x0100; } /* Create a 100 word hole */ }
```

   Now invoke the linker with the --fill_value option:

```
cl55 --run_linker --fill_value=0xFFFF link.cmd
```

   This fills the hole with 0xFFFF.

4. If you do not invoke the linker with the --fill_value option or otherwise specify a fill value, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

9.6.10.4 Explicit Initialization of Uninitialized Sections

You can force the linker to initialize an uninitialized section by specifying an explicit fill value for it in the SECTIONS directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
   .bss: fill = 0x1234 /* Fills .bss with 0x1234 */
}
```

Filling Sections

NOTE: Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large sections or holes.

9.7 Object Libraries

An object library is a partitioned archive file that contains object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as linker input, the linker includes any members of the library that define existing unresolved symbol references. You can use the archiver to build and maintain libraries. Section 8.1 contains more information about the archiver.

Using object libraries can reduce link time and the size of the executable module. Normally, if an object file that contains a function is specified at link time, the file is linked whether the function is used or not; however, if that same function is placed in an archive library, the file is included only if the function is referenced.

The order in which libraries are specified is important, because the linker includes only those members that resolve symbols that are undefined at the time the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. Alternatively, you can use the --reread_libs option to reread libraries until no more references can be resolved (see Section 9.4.12.3). A library has a table that lists all external symbols defined in the library; the linker searches through the table until it determines that it cannot use the library to resolve any more references.

The following examples link several files and libraries, using these assumptions:

- Input files f1.obj and f2.obj both reference an external function named *clrscr*.
- Input file f1.obj references the symbol *origin*.
- Input file f2.obj references the symbol *fillclr*.
- Member 0 of library libc.lib contains a definition of *origin*.
- Member 3 of library liba.lib contains a definition of *fillclr*.
- Member 1 of both libraries defines *clrscr*.

If you enter:

```
cl55 --run_linker f1.obj f2.obj liba.lib libc.lib
```

then:

- Member 1 of liba.lib satisfies the f1.obj and f2.obj references to *clrscr* because the library is searched and the definition of *clrscr* is found.
- Member 0 of libc.lib satisfies the reference to *origin*.
- Member 3 of liba.lib satisfies the reference to *fillclr*.

If, however, you enter:

```
cl55 --run_linker f1.obj f2.obj libc.lib liba.lib
```

then the references to *clrscr* are satisfied by member 1 of libc.lib.

If none of the linked files reference symbols defined in a library, you can use the --undef_sym option to force the linker to include a library member. (See Section 9.4.27.) The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
cl55 --run_linker --undef_sym=rout1 libc.lib
```

If any member of libc.lib defines rout1, the linker includes that member.

Library members are allocated according to the SECTIONS directive default allocation algorithm; see Section 9.6.4.

Section 9.4.12 describes methods for specifying directories that contain object libraries.

9.8 Default Allocation Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections that you choose *not* to specify must still be handled by the linker. The linker uses default algorithms to build and allocate sections within the specifications you supply.

If you do not use the MEMORY and SECTIONS directives, the linker allocates output sections as though the definitions in Example 9-18 were specified.

Example 9-18. Default Allocation for TMS320C55x Devices

```
MEMORY
{
   ROM (RIX)    :  origin = 0100h,    length = 0FEFFh
   VECTOR (RIX) :  origin = 0FFFF00h, length = 0100h
   RAM (RWIX)   :  origin = 010100h,  length = 0FFFFh
}

SECTIONS
{
   .text     > ROM
   .switch   > ROM
   .const    > ROM
   .cinit    > ROM
   .vectors  > VECTOR
   .data     > RAM
   .bss      > RAM
   .sysmem   > RAM
   .stack    > RAM
   .sysstack > RAM
   .cio      > RAM
}
```

All .text input sections are concatenated to form a .text output section in the executable output file, and all .data input sections are combined to form a .data output section.

If you use a SECTIONS directive, the linker performs *no part* of the default allocation. Allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described next in Section 9.8.1.

9.8.1 How the Allocation Algorithm Creates Output Sections

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

Method 1 As the result of a SECTIONS directive definition

Method 2 By combining input sections with the same name into an output section that is not defined in a SECTIONS directive

If an output section is formed as a result of a SECTIONS directive, this definition completely determines the section's contents. (See Section 9.6.4 for examples of how to define an output section's content.)

If an output section is formed by combining input sections not specified by a SECTIONS directive, the linker combines all such input sections that have the same name into an output section with that name. For example, suppose the files f1.obj and f2.obj both contain named sections called Vectors and that the SECTIONS directive does not define an output section for them. The linker combines the two Vectors sections from the input files into a single output section named Vectors, allocates it into memory, and includes it in the output file.

By default, the linker does not display a message when it creates an output section that is not defined in the SECTIONS directive. You can use the --warn_sections linker option (see Section 9.4.28) to cause the linker to display a message when it creates a new output section.

After the linker determines the composition of all output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured. If there is no MEMORY directive, the linker uses the default configuration as shown in Example 9-18. (See Section 9.6.3 for more information on configuring memory.)

9.8.2 Reducing Memory Fragmentation

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. The algorithm comprises these steps:

1. Each output section for which you have supplied a specific binding address is placed in memory at that address.
2. Each output section that is included in a specific, named memory range or that has memory attribute restrictions is allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.
3. Any remaining sections are allocated in the order in which they are defined. Sections not defined in a SECTIONS directive are allocated in the order in which they are encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

The PAGE Option

NOTE: 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.

9.9 Linker-Generated Copy Tables

The linker supports extensions to the link command file syntax that enable the following:

- Make it easier for you to copy objects from load-space to run-space at boot time
- Make it easier for you to manage memory overlays at run time
- Allow you to split GROUPs and output sections that have separate load and run addresses

9.9.1 A Current Boot-Loaded Application Development Process

In some embedded applications, there is a need to copy or download code and/or data from one location to another at boot time before the application actually begins its main execution thread. For example, an application may have its code and/or data in FLASH memory and need to copy it into on-chip memory before the application begins execution.

One way you can develop an application like this is to create a copy table in assembly code that contains three elements for each block of code or data that needs to be moved from FLASH into on-chip memory at boot time:

- The load location (load page id and address)
- The run location (run page id and address)
- The size

The process you follow to develop such an application might look like this:

1. Build the application to produce a .map file that contains the load and run addresses of each section that has a separate load and run placement.
2. Edit the copy table (used by the boot loader) to correct the load and run addresses as well as the size of each block of code or data that needs to be moved at boot time.
3. Build the application again, incorporating the updated copy table.
4. Run the application.

This process puts a heavy burden on you to maintain the copy table (by hand, no less). Each time a piece of code or data is added or removed from the application, you must repeat the process in order to keep the contents of the copy table up to date.

9.9.2 An Alternative Approach

You can avoid some of this maintenance burden by using the LOAD_START(), RUN_START(), and SIZE() operators that are already part of the link command file syntax . For example, instead of building the application to generate a .map file, the link command file can be annotated:

```
SECTIONS
{
   .flashcode: { app_tasks.obj(.text) }
      load = FLASH, run = PMEM,
      LOAD_START(_flash_code_ld_start),
      RUN_START(_flash_code_rn_start),
      SIZE(_flash_code_size)

   ...
}
```

In this example, the LOAD_START(), RUN_START(), and SIZE() operators instruct the linker to create three symbols:

Symbol	Description
_flash_code_ld_start	Load address of .flashcode section
_flash_code_rn_start	Run address of .flashcode section
_flash_code_size	Size of .flashcode section

These symbols can then be referenced from the copy table. The actual data in the copy table will be updated automatically each time the application is linked. This approach removes step 1 of the process described in Section 9.9.1.

While maintenance of the copy table is reduced markedly, you must still carry the burden of keeping the copy table contents in sync with the symbols that are defined in the link command file. Ideally, the linker would generate the boot copy table automatically. This would avoid having to build the application twice *and* free you from having to explicitly manage the contents of the boot copy table.

For more information on the LOAD_START(), RUN_START(), and SIZE() operators, see Section 9.6.9.7.

9.9.3 Overlay Management Example

Consider an application which contains a memory overlay that must be managed at run time. The memory overlay is defined using a UNION in the link command file as illustrated in Example 9-19:

Example 9-19. Using a UNION for Memory Overlay

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1: { task1.obj(.text) }
         .task2: { task2.obj(.text) }

      } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)

      GROUP
      {
         .task3: { task3.obj(.text) }
         .task4: { task4.obj(.text) }

      } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)

   } run = RAM, RUN_START(_task_run_start)

 ...
}
```

The application must manage the contents of the memory overlay at run time. That is, whenever any services from .task1 or .task2 are needed, the application must first ensure that .task1 and .task2 are resident in the memory overlay. Similarly for .task3 and .task4.

To affect a copy of .task1 and .task2 from ROM to RAM at run time, the application must first gain access to the load address of the tasks (_task12_load_start), the run address (_task_run_start), and the size (_task12_size). Then this information is used to perform the actual code copy.

9.9.4 Generating Copy Tables Automatically With the Linker

The linker supports extensions to the link command file syntax that enable you to do the following:

- Identify any object components that may need to be copied from load space to run space at some point during the run of an application
- Instruct the linker to automatically generate a copy table that contains (at least) the load address, run address, and size of the component that needs to be copied
- Instruct the linker to generate a symbol specified by you that provides the address of a linker-generated copy table. For instance, Example 9-19 can be written as shown in Example 9-20:

Example 9-20. Produce Address for Linker Generated Copy Table

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1: { task1.obj(.text) }
         .task2: { task2.obj(.text) }

      } load = ROM, table(_task12_copy_table)

      GROUP
      {
         .task3: { task3.obj(.text) }
         .task4: { task4.obj(.text) }

      } load = ROM, table(_task34_copy_table)

   } run = RAM
   ...
}
```

Using the SECTIONS directive from Example 9-20 in the link command file, the linker generates two copy tables named: _task12_copy_table and _task34_copy_table. Each copy table provides the load address, run address, and size of the GROUP that is associated with the copy table. This information is accessible from application source code using the linker-generated symbols, _task12_copy_table and _task34_copy_table, which provide the addresses of the two copy tables, respectively.

Using this method, you do not have to worry about the creation or maintenance of a copy table. You can reference the address of any copy table generated by the linker in C/C++ or assembly source code, passing that value to a general purpose copy routine which will process the copy table and affect the actual copy.

9.9.5 The table() Operator

You can use the table() operator to instruct the linker to produce a copy table. A table() operator can be applied to an output section, a GROUP, or a UNION member. The copy table generated for a particular table() specification can be accessed through a symbol specified by you that is provided as an argument to the table() operator. The linker creates a symbol with this name and assigns it the address of the copy table as the value of the symbol. The copy table can then be accessed from the application using the linker-generated symbol.

Each table() specification you apply to members of a given UNION must contain a unique name. If a table() operator is applied to a GROUP, then none of that GROUP's members may be marked with a table() specification. The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table() specification. The linker does not generate a copy table for erroneous table() operator specifications.

Copy tables can be generated automatically; see Section 9.9.4.

9.9.6 Boot-Time Copy Tables

The linker supports a special copy table name, BINIT (or binit), that you can use to create a boot-time copy table. For example, the link command file for the boot-loaded application described in Section 9.9.2 can be rewritten as follows:

```
SECTIONS
{
   .flashcode: { app_tasks.obj(.text) }
   load = FLASH, run = PMEM,
      table(BINIT)
   ...
}
```

For this example, the linker creates a copy table that can be accessed through a special linker-generated symbol, __binit__, which contains the list of all object components that need to be copied from their load location to their run location at boot-time. If a link command file does not contain any uses of table(BINIT), then the __binit__ symbol is given a value of -1 to indicate that a boot-time copy table does not exist for a particular application.

You can apply the table(BINIT) specification to an output section, GROUP, or UNION member. If used in the context of a UNION, only one member of the UNION can be designated with table(BINIT). If applied to a GROUP, then none of that GROUP's members may be marked with table(BINIT).The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table(BINIT) specification.

9.9.7 Using the table() Operator to Manage Object Components

If you have several pieces of code that need to be managed together, then you can apply the same table() operator to several different object components. In addition, if you want to manage a particular object component in multiple ways, you can apply more than one table() operator to it. Consider the link command file excerpt in Example 9-21:

Example 9-21. Linker Command File to Manage Object Components

```
SECTIONS
{
   UNION
   {
      .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second: { a2.obj(.text), b2.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra: load = EMEM, run = PMEM, table(BINIT)
   ...
}
```

In this example, the output sections .first and .extra are copied from external memory (EMEM) into program memory (PMEM) at boot time while processing the BINIT copy table. After the application has started executing its main thread, it can then manage the contents of the overlay using the two overlay copy tables named: _first_ctbl and _second_ctbl.

9.9.8 Compression Support

When automatically generating copy tables, the linker provides a way to compress the load-space data. This can reduce the read-only memory foot print. This compressed data can be decompressed while copying the data from load space to run space.

When you choose compression, it is not guaranteed that the linker will compress the load data. The linker compresses load data only when such compression reduces the overall size of the load space. In some cases even if the compression results in smaller load section size the linker does not compress the data if the decompression routine offsets for the savings.

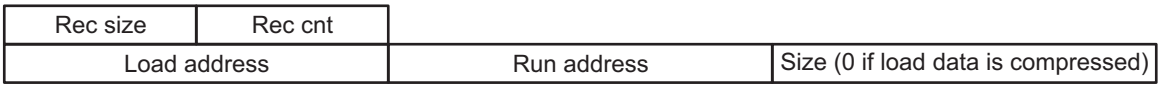
For example, assume RLE compression reduces the size of section1 by 30 bytes. Also assume the RLE decompression routine takes up 40 bytes in load space. By choosing to compress section1 the load space is increased by 10 bytes. Therefore, the linker will not compress section1. On the other hand, if there is another section (say section2) that can benefit by more than 10 bytes from applying the same compression then both sections can be compressed and the overall load space is reduced. In such cases the linker compresses both the sections.

You cannot force the linker to compress the data when doing so does not result in savings.

9.9.8.1 Compressed Copy Table Format

The copy table format is the same irrespective of the compression. The size field of the copy record is overloaded to support compression. Figure 9-7 illustrates the compressed copy table layout.

Figure 9-7. Compressed Copy Table



In Figure 9-7, if the size in the copy record is non-zero it represents the size of the data to be copied, and also means that the size of the load data is the same as the run data. When the size is 0, it means that the load data is compressed.

9.9.8.2 Compressed Section Representation in the Object File

When the load data is not compressed, the object file can have only one section with a different load and run address.

Consider the following table() operation in the linker command file.

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table)
}
```

The output object file has one output section named .task1 which has a different load and run addresses. This is possible because the load space and run space have identical data when the section is not compressed.

Alternatively, consider the following:

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table, compression=rle)
}
```

If the linker compresses the .task1 section then the load space data and the run space data are different. The linker creates the following two sections:

- **.task1** : This section is uninitialized. This output section represents the run space image of section task1.
- **.task1.load** : This section is initialized. This output section represents the load space image of the section task1. This section usually is considerably smaller in size than .task1 output section.

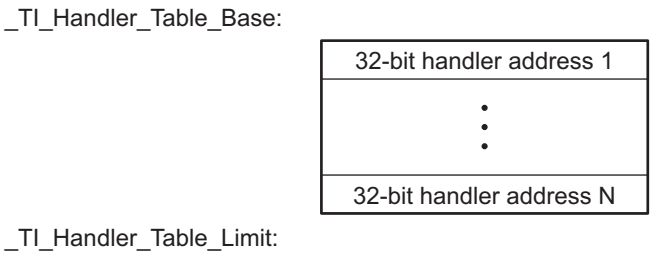
9.9.8.3 Compressed Data Layout

The compressed load data has the following layout:

8-bit index	Compressed data

The first eight bits of the load data are the handler index. This handler index is used to index into a handler table to get the address of a handler function that knows how to decode the data that follows. The handler table is a list of 32-bit function pointers as shown in Figure 9-8.

Figure 9-8. Handler Table



The linker creates a separate output section for the load and run space. For example, if .task1.load is compressed using RLE, the handler index points to an entry in the handler table that has the address of the run-time-support routine __TI_decompress_rle().

9.9.8.4 Run-Time Decompression

During run time you call the run-time-support routine copy_in() to copy the data from load space to run space. The address of the copy table is passed to this routine. First the routine reads the record count. Then it repeats the following steps for each record:

1. Read load address, run address and size from record.
2. If size is zero go to step 5.
3. Call memcpy passing the run address, load address and size.
4. Go to step 1 if there are more records to read.
5. Read the first byte from load address. Call this index.
6. Read the handler address from (&__TI_Handler_Base)[index].
7. Call the handler and pass load address + 1 and run address.
8. Go to step 1 if there are more records to read.

The routines to handle the decompression of load data are provided in the run-time-support library.

9.9.8.5 Compression Algorithms

Run Length Encoding (RLE):

8-bit index	Initialization data compressed using run length encoding

The data following the 8-bit index is compressed using run length encoded (RLE) format. uses a simple run length encoding that can be decompressed using the following algorithm:

1. Read the first byte, Delimiter (D).
2. Read the next byte (B).
3. If B != D, copy B to the output buffer and go to step 2.
4. Read the next byte (L).
   (a) If L == 0, then length is either a 16-bit, a 24-bit value, or we've reached the end of the data, read next byte (L).
      (i) If L == 0, length is a 24-bit value or the end of the data is reached, read next byte (L).
         (i) If L == 0, the end of the data is reached, go to step 7.
         (ii) Else L <<= 16, read next two bytes into lower 16 bits of L to complete 24-bit value for L.
      (ii) Else L <<= 8, read next byte into lower 8 bits of L to complete 16-bit value for L.
   (b) Else if L > 0 and L < 4, copy D to the output buffer L times. Go to step 2.
   (c) Else, length is 8-bit value (L).
5. Read the next byte (C); C is the repeat character.
6. Write C to the output buffer L times; go to step 2.
7. End of processing.

The run-time support library has a routine __TI_decompress_rle24() to decompress data compressed using RLE. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

RLE Decompression Routine

NOTE: The previous decompression routine, __TI_decompress_rle(), is included in the run-time-support library for decompressing RLE encodings that are generated by older versions of the linker.

Lempel-Ziv-Storer-Szymanski Compression (LZSS):

8-bit index	Data compressed using LZSS

The data following the 8-bit index is compressed using LZSS compression. The run-time-support library has the routine __TI_decompress_lzss() to decompress the data compressed using LZSS. The first argument to this function is the address pointing to the byte after the 8-bit Index, and the second argument is the run address from the C auto initialization record.

9.9.9 Copy Table Contents

In order to use a copy table that is generated by the linker, you must be aware of the contents of the copy table. This information is included in a new run-time-support library header file, cpy_tbl.h, which contains a C source representation of the copy table data structure that is automatically generated by the linker.

Example 9-22 shows the TMS320C55x copy table header file.

Example 9-22. TMS320C55x cpy_tbl.h File

```
/*****************************************************************************/
/* cpy_tbl.h vxxxxx                                                          */
/* Copyright (c) 2003 Texas Instruments Incorporated                         */
/*                                                                           */
/* Specification of copy table data structures which can be automatically    */
/* generated by the linker (using the table() operator in the LCF).          */
/*****************************************************************************/
#ifndef _CPY_TBL
#define _CPY_TBL

#include <stdlib.h>

/*****************************************************************************/
/* Copy Record Data Structure                                                */
/*****************************************************************************/
typedef struct copy_record
{
   unsigned long load_loc;
   unsigned long run_loc;
   unsigned long size;
} COPY_RECORD;

/*****************************************************************************/
/* Copy Table Data Structure                                                 */
/*****************************************************************************/
typedef struct copy_table
{
   unsigned short rec_size;
   unsigned short num_recs;
   COPY_RECORD  recs[1];
} COPY_TABLE;

/*****************************************************************************/
/* Prototype for general purpose copy routine.                               */
/*****************************************************************************/
extern void copy_in(COPY_TABLE *tp);

/*****************************************************************************/
/* Prototypes for I/O aware copy routines used in copy_in().                 */
/*****************************************************************************/
extern void cpy_io_to_io(void *from, void *to, size_t n);
extern void cpy_io_to_mem(void *from, void *to, size_t n);
extern void cpy_mem_to_io(void *from, void *to, size_t n);

#endif /* ! _CPY_TBL */
```

For each object component that is marked for a copy, the linker creates a COPY_RECORD object for it. Each COPY_RECORD contains at least the following information for the object component:

- The load page id
- The run page id
- The load address
- The run address
- The size

The linker collects all COPY_RECORDs that are associated with the same copy table into a COPY_TABLE object. The COPY_TABLE object contains the size of a given COPY_RECORD, the number of COPY_RECORDs in the table, and the array of COPY_RECORDs in the table. For instance, in the BINIT example in Section 9.9.6, the .first and .extra output sections will each have their own COPY_RECORD entries in the BINIT copy table. The BINIT copy table will then look like this:

```
COPY_TABLE __binit__ = { 12, 2,
                         { <load page id and address of .first>,
                           <run page id and address of .first>,
                           <size of .first> },
                         { <load page id and address of .extra>,
                           <run page id and address of .extra>,
                           <size of .extra> } };
```

9.9.10 General Purpose Copy Routine

The cpy_tbl.h file in Example 9-22 also contains a prototype for a general-purpose copy routine, copy_in(), which is provided as part of the run-time-support library. The copy_in() routine takes a single argument: the address of a linker-generated copy table. The routine then processes the copy table data object and performs the copy of each object component specified in the copy table.

The copy_in() function definition is provided in the cpy_tbl.c run-time-support source file shown in Example 9-23.

Example 9-23. Run-Time-Support cpy_tbl.c File

```
/****************************************************************************/
/* cpy_tbl.c                                                                */
/*                                                                          */
/* Copyright (c) 2003 Texas Instruments Incorporated                        */
/*                                                                          */
/* General purpose copy routine. Given the address of a linker-generated    */
/* COPY_TABLE data structure, effect the copy of all object components      */
/* that are designated for copy via the corresponding LCF table() operator. */
/*                                                                          */
/****************************************************************************/
#include <cpy_tbl.h>
#include <string.h>

/****************************************************************************/
/* Static Function Prototypes for I/O aware copy routines.                  */
/****************************************************************************/
static void cpy_io_to_io(void *from, void *to, size_t n);
static void cpy_io_to_mem(void *from, void *to, size_t n);
static void cpy_mem_to_io(void *from, void *to, size_t n);

/****************************************************************************/
/* COPY_IN()                                                                */
/****************************************************************************/
void copy_in(COPY_TABLE *tp)
{
   unsigned short i;
   for (i = 0; i < tp->num_recs; i++)
   {
      COPY_RECORD   *crp       = &tp->recs[i];
```

Example 9-23. Run-Time-Support cpy_tbl.c File (continued)

```
      int           load_pgid  = (int)(crp->load_loc >> 24);
      unsigned char *load_addr = (unsigned char *)(crp->load_loc & 0x7fffff);
      int           run_pgid   = (int)(crp->run_loc >> 24);

      unsigned char *run_addr  = (unsigned char *)(crp->run_loc & 0x7fffff);
      unsigned int  cpy_type   = 0;

      /*********************************************************************/
      /* If page ID != 0, location is assumed to be in I/O memory.         */
      /*********************************************************************/
      if (load_pgid) cpy_type += 2;
      if (run_pgid) cpy_type += 1;

      /*********************************************************************/
      /* Dispatch to appropriate copy routine based on whether or not load  */
      /* and/or run location is in I/O memory.                              */
      /*********************************************************************/
     switch (cpy_type)
     {
        case 3: cpy_io_to_io(load_addr, run_addr, crp->size) ; break;
        case 2: cpy_io_to_mem(load_addr, run_addr, crp->size); break;
        case 1: cpy_mem_to_io(load_addr, run_addr, crp->size); break;
        case 0: memcpy(run_addr, load_addr, crp->size); break;
     }
   }
}
/*************************************************************************/
/* CPY_IO_TO_IO() - Move code/data from one location in I/O to another.  */
/*************************************************************************/
static void cpy_io_to_io(void *from, void *to, size_t n)
{
   ioport unsigned char *src = (unsigned char *)from;
   ioport unsigned char *dst = (unsigned char *)to;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *dst++ = *src++;
}

/*************************************************************************/
/* CPY_IO_TO_MEM() - Move code/data from I/O to normal system memory.    */
/*************************************************************************/
static void cpy_io_to_mem(void *from, void *to, size_t n)
{
   ioport unsigned char *src = (unsigned char *)from;
   unsigned char        *dst = (unsigned char *)to;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *dst++ = *src++;
}

/*************************************************************************/
/* CPY_MEM_TO_IO() - Move code/data from normal memory to I/O.           */
/*************************************************************************/
static void cpy_mem_to_io(void *from, void *to, size_t n)
{
   unsigned char        *src = (unsigned char *)from;
   ioport unsigned char *dst = (unsigned char *)to;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *dst++ = *src++;
}
```

The load and run page id's are unpacked from the load_loc and run_loc fields and used to select the appropriate subroutine for copying from the source memory type to the destination memory type. A page id of 0 indicates that the specified address is in normal C55x memory, and a non-zero page id indicates that the address is in I/O memory. The general-purpose copy routine utilizes special copy routines if the code/data needs to be moved into and/or out of I/O memory.

A pointer can be qualified with the ioport keyword to indicate that any memory reads from that address need to be qualified with a readport() instruction modifier or a port() operand modifier. Likewise, a memory write to such a pointer needs to be qualified with a writeport() instruction modifier or a port() operand modifier. By qualifying the pointer with the ioport keyword, the compiler generates these I/O modifiers automatically.

9.9.11 Linker-Generated Copy Table Sections and Symbols

The linker creates and allocates a separate input section for each copy table that it generates. Each copy table symbol is defined with the address value of the input section that contains the corresponding copy table.

The linker generates a unique name for each overlay copy table input section. For example, table(_first_ctbl) would place the copy table for the .first section into an input section called .ovly:_first_ctbl. The linker creates a single input section, .binit, to contain the entire boot-time copy table.

Example 9-24 illustrates how you can control the placement of the linker-generated copy table sections using the input section names in the link command file.

Example 9-24. Controlling the Placement of the Linker-Generated Copy Table Sections

```
SECTIONS
{
   UNION
   {
      .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
              load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second: { a2.obj(.text), b2.obj(.text) }
              load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra: load = EMEM, run = PMEM, table(BINIT)

   ...

   .ovly: { } > BMEM
   .binit: { } > BMEM
}
```

For the link command file in Example 9-24, the boot-time copy table is generated into a .binit input section, which is collected into the .binit output section, which is mapped to an address in the BMEM memory area. The _first_ctbl is generated into the .ovly:_first_ctbl input section and the _second_ctbl is generated into the .ovly:_second_ctbl input section. Since the base names of these input sections match the name of the .ovly output section, the input sections are collected into the .ovly output section, which is then mapped to an address in the BMEM memory area.

If you do not provide explicit placement instructions for the linker-generated copy table sections, they are allocated according to the linker's default placement algorithm.

The linker does not allow other types of input sections to be combined with a copy table input section in the same output section. The linker does not allow a copy table section that was created from a partial link session to be used as input to a succeeding link session.

9.9.12 Splitting Object Components and Overlay Management

In previous versions of the linker, splitting sections that have separate load and run placement instructions was not permitted. This restriction was because there was no effective mechanism for you, the developer, to gain access to the load address or run address of each one of the pieces of the split object component. Therefore, there was no effective way to write a copy routine that could move the split section from its load location to its run location.

However, the linker can access both the load address and run address of every piece of a split object component. Using the table() operator, you can tell the linker to generate this information into a copy table. The linker gives each piece of the split object component a COPY_RECORD entry in the copy table object.

For example, consider an application which has seven tasks. Tasks 1 through 3 are overlaid with tasks 4 through 7 (using a UNION directive). The load placement of all of the tasks is split among four different memory areas (LMEM1, LMEM2, LMEM3, and LMEM4). The overlay is defined as part of memory area PMEM. You must move each set of tasks into the overlay at run time before any services from the set are used.

You can use table() operators in combination with splitting operators, >>, to create copy tables that have all the information needed to move either group of tasks into the memory overlay as shown in Example 9-25. Example 9-26 illustrates a possible driver for such an application.

Example 9-25. Creating a Copy Table to Access a Split Object Component

```
SECTIONS
{
   UNION
   {
      .task1to3: { *(.task1), *(.task2), *(.task3) }
              load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)

      GROUP
      {
         .task4: { *(.task4) }
         .task5: { *(.task5) }
         .task6: { *(.task6) }
         .task7: { *(.task7) }

      } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)

   } run = PMEM

   ...

   .ovly: > LMEM4
}
```

Example 9-26. Split Object Component Driver

```
#include <cpy_tbl.h>

extern far COPY_TABLE task13_ctbl;
extern far COPY_TABLE task47_ctbl;

extern void task1(void);
...
extern void task7(void);

main()
{
   ...
   copy_in(&task13_ctbl);
   task1();
   task2();
   task3();
   ...

   copy_in(&task47_ctbl);
   task4();
   task5();
   task6();
   task7();
   ...
}
```

You must declare a COPY_TABLE object as *far* to allow the overlay copy table section placement to be independent from the other sections containing data objects (such as .bss).

The contents of the .task1to3 section are split in the section's load space and contiguous in its run space. The linker-generated copy table, _task13_ctbl, contains a separate COPY_RECORD for each piece of the split section .task1to3. When the address of _task13_ctbl is passed to copy_in(), each piece of .task1to3 is copied from its load location into the run location.

The contents of the GROUP containing tasks 4 through 7 are also split in load space. The linker performs the GROUP split by applying the split operator to each member of the GROUP in order. The copy table for the GROUP then contains a COPY_RECORD entry for every piece of every member of the GROUP. These pieces are copied into the memory overlay when the _task47_ctbl is processed by copy_in().

The split operator can be applied to an output section, GROUP, or the load placement of a UNION or UNION member. The linker does not permit a split operator to be applied to the run placement of either a UNION or of a UNION member. The linker detects such violations, emits a warning, and ignores the offending split operator usage.

9.10 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as *partial linking* or *incremental linking*. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program.

Follow these guidelines for producing a file that you will relink:

- The intermediate files produced by the linker *must* have relocation information. Use the --relocatable option when you link the file the first time. (See Section 9.4.2.2.)
- Intermediate files *must* have symbolic information. By default, the linker retains symbolic information in its output. Do not use the --no_sym_table option if you plan to relink a file, because --no_sym_table strips symbolic information from the output module. (See Section 9.4.18.)
- Intermediate link operations 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.
- If the intermediate files have global symbols that have the same name as global symbols in other files and you want them to be treated as static (visible only within the intermediate file), you must link the files with the --make_static option (see Section 9.4.13.1).
- If you are linking C code, do not use --ram_model or --rom_model until the final linker. Every time you invoke the linker with the --ram_model or --rom_model option, the linker attempts to create an entry point. (See Section 9.4.20.)

The following example shows how you can use partial linking:

Step 1: Link the file file1.com; use the --relocatable option to retain relocation information in the output file tempout1.out.

```
cl55 --run_linker --relocatable --output_file=tempout1 file1.com
```

file1.com contains:

```
SECTIONS
{
    ss1:   {
           f1.obj
           f2.obj
            .
            .
            .
           fn.obj
           }
}
```

Step 2: Link the file file2.com; use the --relocatable option to retain relocation information in the output file tempout2.out.

```
cl55 --run_linker --relocatable --output_file=tempout2 file2.com
```

file2.com contains:

```
SECTIONS
{
   ss2:   {
           g1.obj
           g2.obj
            .
            .
            .
           gn.obj
           }
}
```

Step 3: Link tempout1.out and tempout2.out.

```
cl55 --run_linker --map_file=final.map --output_file=final.out tempout1.out
tempout2.out
```

9.11 Linking C/C++ Code

The C/C++ compiler produces assembly language source code that can be assembled and linked. For example, a C program consisting of modules prog1, prog2, etc., can be assembled and then linked to produce an executable file called prog.out:

```
cl55 --run_linker --rom_model --output_file prog.out prog1.obj prog2.obj ... rts55.lib
```

The --rom_model option tells the linker to use special conventions that are defined by the C/C++ environment.

The archive libraries shipped by TI contain C/C++ run-time-support functions.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

For more information about the TMS320C55x C/C++ language, including the run-time environment and run-time-support functions, see the *TMS320C55x Optimizing C/C++ Compiler User's Guide*.

9.11.1 Run-Time Initialization

All C/C++ programs must be linked with code to initialize and execute the program, called a *bootstrap* routine, also known as the *boot.obj* object module. The symbol _c_int00 is defined as the program entry point and is the start of the C boot routine in boot.obj; referencing _c_int00 ensures that boot.obj is automatically linked in from the run-time-support library. When a program begins running, it executes boot.obj first. The boot.obj symbol contains code and data for initializing the run-time environment and performs the following tasks:

- Sets up the system stack
- Sets up the primary and secondary system stacks
- Processes the run-time *.cinit* initialization table and autoinitializes global variables (when the linker is invoked with the --rom_model option)

The run-time-support object libraries contain boot.obj. You can:

- Use the archiver to extract boot.obj from the library and then link the module in directly.
- Include the appropriate run-time-support library as an input file (the linker automatically extracts boot.obj when you use the --ram_model or --rom_model option).

9.11.2 Object Libraries and Run-Time Support

The *TMS320C55x Optimizing C/C++ Compiler User's Guide* describes additional run-time-support functions that are included in rts.src. If your program uses any of these functions, you must link the appropriate run-time-support library with your object files.

You can also create your own object libraries and link them. The linker includes and links only those library members that resolve undefined references.

9.11.3 Setting the Size of the Stack and Heap Sections

The C/C++ language uses three uninitialized sections called .sysmem, .stack, and .sysstack for the memory pool used by the malloc( ) functions and the run-time stacks, respectively. You can set the size of these by using the --heap_size, --stack_size, or --sysstack option and specifying the size of the section as a 4-byte constant immediately after the option. If the options are not used, the default size of the heap is 2K bytes and the default size of the stack is 1K bytes.

Allocation of .stack and .sysstack Sections

NOTE: The .stack and .sysstack sections must be allocated on the same 64K-word data page.

See Section 9.4.10 for setting heap sizes, Section 9.4.23 for setting system stack sizes, and Section 9.4.26 for setting secondary stack sizes.

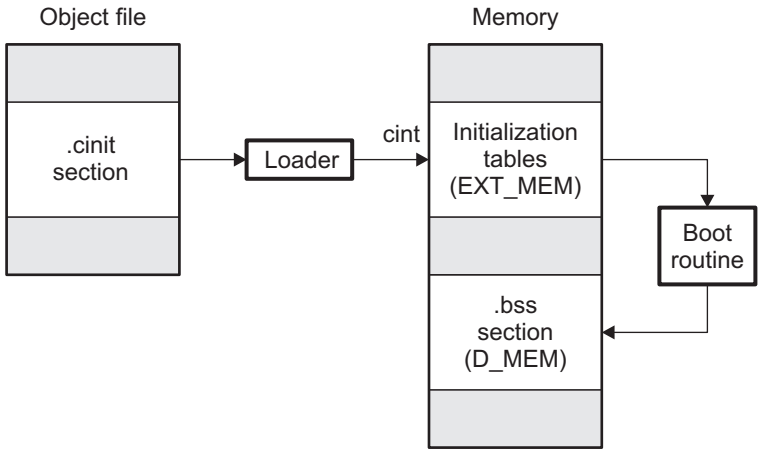
9.11.4 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in slow external memory and copied to fast external memory each time the program starts.

Figure 9-9 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into slow external memory.

Figure 9-9. Autoinitialization at Run Time



9.11.5 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

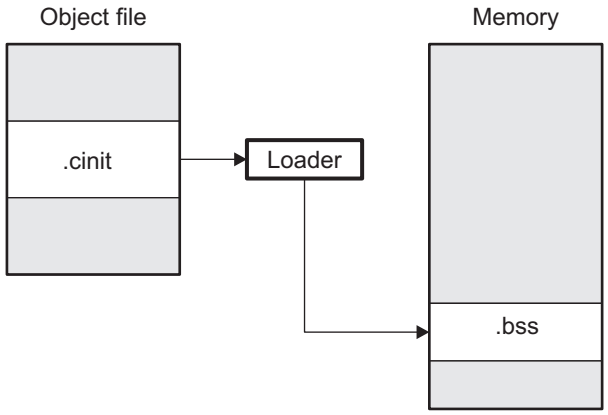
When you use the --ram_model linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to -1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the .cinit section in the object file.
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory.
- Understand the format of the initialization tables.

Figure 9-10 illustrates the initialization of variables at load time.

Figure 9-10. Initialization at Load Time



9.11.6 The --rom_model and --ram_model Linker Options

The following list outlines what happens when you invoke the linker with the --ram_model or --rom_model option.

- The symbol _c_int00 is defined as the program entry point. The _c_int00 symbol is the start of the C boot routine in boot.obj; referencing _c_int00 ensures that boot.obj is automatically linked in from the appropriate run-time-support library.
- The .cinit output section is padded with a termination record to designate to the boot routine (autoinitialize at run time) or the loader (initialize at load time) when to stop reading the initialization tables.
- When you initialize at load time (--ram_model option):
  - The linker sets cinit to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag (0010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform initialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.
- When you autoinitialize at run time (--rom_model option), the linker defines cinit as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.

9.12 Linker Example

This example links three object files named demo.obj, ctrl.obj, and tables.obj and creates a program called demo.out.

Assume that target memory has the following program memory configuration:

Program Memory

Address Range	Contents
0x0080 to 0x7000	On-chip RAM_PG
0xC000 to 0xFF80	On-chip ROM

Data Memory

Address Range	Contents
0x0080 to 0x0FFF	RAM block ONCHIP
0x0060 to 0xFFFF	Mapped external addresses EXT

Byte Address Range	Contents
000100 to 007080	On-chip RAM_PG
007081 to 008000	RAM block ONCHIP
008001 to 00A000	Mapped external addresses EXT
00C000 to 00FF80	On-chip ROM

The output sections are constructed in the following manner:

- Executable code, contained in the .text sections of demo.obj, fft.obj, and tables.obj, is linked into program memory ROM.
- Variables, contained in the var_defs section of demo.obj, are linked into data memory in block FAST_MEM_2.
- Tables of coefficients in the .data sections of demo.obj, tables.obj, and fft.obj are linked into FAST_MEM_1. A hole is created with a length of 100 and a fill value of 0x07A1C.
- The xy section form demo.obj, which contains buffers and variables, is linked by default into page 1 of the block STACK, since it is not explicitly linked.
- 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 bytes 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, has the default linking into block ONCHIP of data RAM, since it was not explicitly linked.

Example 9-27 shows the link command file for this example. Example 9-28 shows the map file.

Example 9-27. Linker Command File, demo.cmd

```
/*************************************************************/
/***              Specify Linker Options                   ***/
/*************************************************************/
--entry_point coeff          /* Define the program entry point */
--output_file=demo.out       /* Name the output file           */
--map_file=demo.map          /* Create an output map           */

/*************************************************************/
/***              Specify the Input Files                  ***/
/*************************************************************/

demo.obj
fft.obj
tables.obj

/*************************************************************/
/***            Specify the Memory Configurations          ***/
/*************************************************************/

MEMORY
{
   RAM_PG: origin=00100h   length=06F80h
   ONCHIP: origin=007081h  length=0F7Fh
   EXT:    origin=08001h   length=01FFFh
   ROM:    origin=0C000h   length=03F80h
}

/*************************************************************/
/***              Specify the Output Sections              ***/
/*************************************************************/

SECTIONS
{
        .text: load = ROM               /* link .text into ROM */
        var_defs: load = ONCHIP                 /* defs in RAM */
        .data: fill = 07A1Ch, load=ONCHIP

        {
               tables.obj(.data) /* .data input */
               fft.obj(.data)    /* .data input */
               . = 100h;      /* create hole, fill with 07A1Ch */
        }                               /* and link with ONCHIP */
        .bss: load=RAM_PG,fill=0FFFFh
                               /* Remaining .bss; fill and link */
}
/*************************************************************/
/***                   End of Command File                 ***/
/*************************************************************/
```

Invoke the linker by entering the following command:

```
cl55 --run_linker demo.cmd
```

This creates the map file shown in Example 9-28 and an output file called demo.out that can be run on a TMS320C55x.

Example 9-28. Output Map File, demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: 0

MEMORY CONFIGURATION
  name    org(bytes) len(bytes) used(bytes) attributes fill
 --------  --------  ---------  ---------  ----------  --------
 RAM_PG    00000100  000006f80   00000064    RWIX
 ONCHIP    00007081  000000f7f   00000104    RWIX
 EXT       00008000  000001fff   00000000    RWIX
 ROM       0000c000  000003f80   0000001f    RWIX

SECTION ALLOCATION MAP
 output                                                  attributes/
section  page  org(bytes) org(words) len(bytes) len(words) input sections
-------- ---- ---------- ---------- ---------- ---------- ----------------
.text     0    0000c000              0000001f
               0000c000              0000000a              tables.obj(.text)
               0000c00a              00000008              fft.obj (.text)
               0000c012              0000000c              demo.obj (.text)
               0000c01e              00000001              --HOLE-- [fill = 2020]
var_defs  0               00003841              00000002
                          00003841              00000002   fft.obj (var_defs)
.data     0               00003843              00000080
                          00003843              00000001   tables.obj (.data)
                          00003844              00000004   fft.obj (.data)
                          00003848              0000007b   --HOLE-- [fill = 7a1c]
                          000038c3              00000000   demo.obj (.data)
.bss      0               00000080              00000002
                          00000080              00000002   demo.obj(.bss)[fill=ffff]
                          00000082              00000000   fft.obj (.bss)
                          00000082              00000000   tables.obj (.bss)
xy        0               00000082              00000030   UNINITIALIZED
                          00000082              00000030   demo.obj (xy)

GLOBAL SYMBOLS:
Sorted alphabetically by name              Sorted by symbol address
abs. value/                              abs. value/
byte addr  word addr  name               byte addr  word addr  name
---------  ---------  -----              --------   --------  ----
           00000080   .bss                          00000080   .bss
           00003843   .data                         00000082   end
0000c000              .text                         00003843   .data
0000c016              ARRAY                         00003843   TEMP
           00003843   TEMP                          000038c3   edata
0000c012              _x42               0000c012              _x42
           000038c3   edata              0000c000              .text
           00000082   end                0000c016              ARRAY
0000c01f              etext              0000c01f              etext
```

Absolute Lister Description

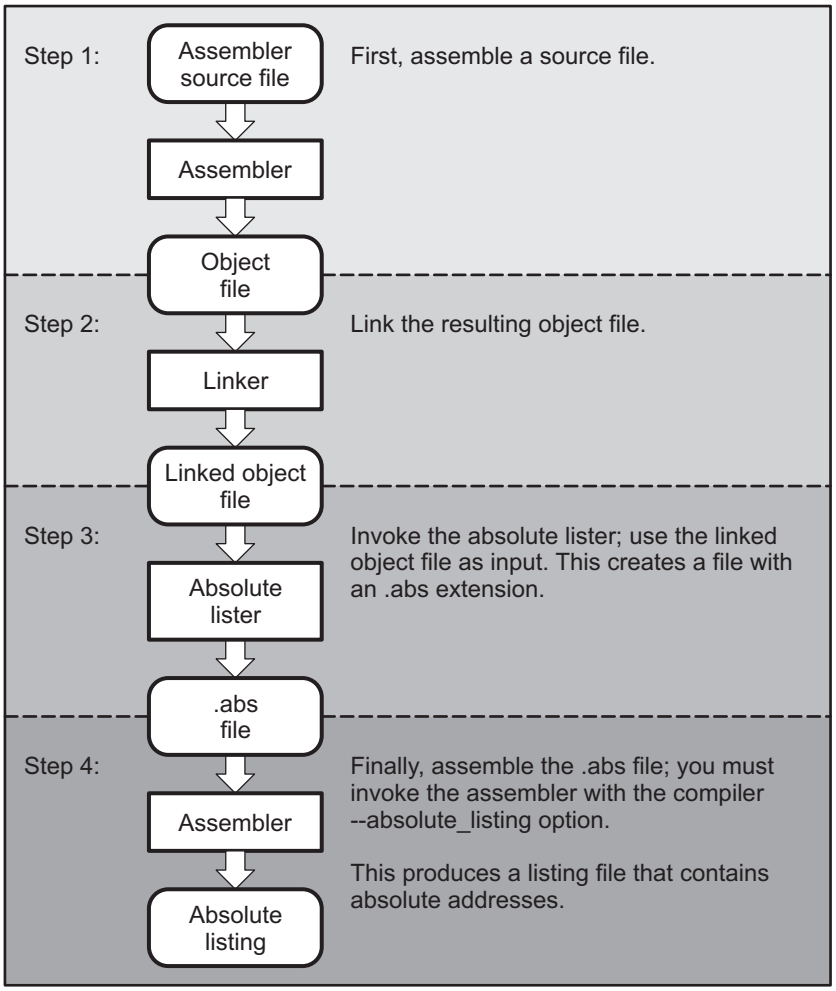
The TMS320C55x absolute lister is a debugging tool that accepts linked object files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code. Manually, this could be a tedious process requiring many operations; 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:

abs55 [-*options*] *input file*

abs55 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 valid options are:

- 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)
- ep [.]*pext* for CPP source files (default is cpp)

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 prompts you for one.

The absolute lister produces an output file for each file that was linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

Assemble these files with the --absolute_listing assembler option as follows to create the absolute listing:

cl55 --absolute_listing *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 (--symdebug:dwarf compiler option). When the debugging option is set, the resulting linked object file contains the name of the source files used to build it. In this case, the absolute lister does not generate a corresponding .abs file for the C header files. Also, the .abs file corresponding to a C source file uses the assembly file generated from the C source file rather than the C source file itself.

For example, suppose the C source file hello.csr is compiled with the debugging option set; the debugging option generates the assembly file hello.s. The hello.csr file includes hello.hsr. Assuming the executable file created is called hello.out, the following command generates the proper .abs file:

```
abs55 -ea s -ec csr -eh hsr hello.out
```

An .abs file is not created for hello.hsr (the header file), and hello.abs includes the assembly file hello.s, not the C source file hello.csr.

10.3 Absolute Lister Example

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

module1.asm

```
.bss    array,100
.bss    dflag, 2
.copy   globals.def
.text
MOV     #offset,AC0
MOV     dflag,AC0
```

module2.asm

```
.bss    offset, 2
.copy   globals.def
.text
MOV     #offset,AC0
MOV     #array,AC0
```

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**:

```
cl55 module1
cl55 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:

```
--output_file=bttest.out
--map_file=bttest.map
module1.obj
module2.obj
MEMORY
{
        ROM:   origin=2000h      length=2000h
        RAM:   origin=8000h      length=8000h
}
SECTIONS
{
        .data: >RAM
        .text: >ROM
        .bss:  >RAM
}
```

Invoke the linker:

```
cl55 --run_linker bttest.cmd
```

This command creates an executable object file called bttest.out; use this new file as input for the absolute lister.

Step 3: Now, invoke the absolute lister:

```
abs55 bttest.out
```

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

module1.abs:

```
          .nolist
array     .setsym     0004000h
dflag     .setsym     0004064h
offset    .setsym     0004066h
.data     .setsym     0004000h
__data_   .setsym     0004000h
edata     .setsym     0004000h
__edata_  .setsym     0004000h
.text     .setsym     0002000h
__text_   .setsym     0002000h
etext     .setsym     000200fh
__etext_  .setsym     000200fh
.bss      .setsym     0004000h
__bss_    .setsym     0004000h
end       .setsym     0004068h
__end_    .setsym     0004068h
          .setsect    ".text",0002000h
          .setsect    ".data",0004000h
          .setsect    ".bss",0004000h
          .list
          .text
          .copy       "module1.asm"
```

module2.abs:

```
          .nolist
array     .setsym     0004000h
dflag     .setsym     0004064h
offset    .setsym     0004066h
.data     .setsym     0004000h
__data_   .setsym     0004000h
edata     .setsym     0004000h
__edata_  .setsym     0004000h
.text     .setsym     0002000h
__text_   .setsym     0002000h
etext     .setsym     000200fh
__etext_  .setsym     000200fh
.bss      .setsym     0004000h
__bss_    .setsym     0004000h
end       .setsym     0004068h
__end_    .setsym     0004068h
          .setsect    ".text",0002006h
          .setsect    ".data",0004000h
          .setsect    ".bss",0004066h
          .list
          .text
          .copy       "module2.asm"
```

These files contain the following information that the assembler needs for Step 4:

- They contain .setsym directives, which equate values to global symbols. Both files contain global equates for the symbol *dflag*. The symbol dflag was defined in the file globals.def, which was included in module1.asm and module2.asm.
- They contain .setsect directives, which define the absolute addresses for sections.
- They contain .copy directives, which defines the assembly language source file to include.

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

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

```
cl55 --absolute_listing module1.abs
cl55 --absolute_listing module2.abs
```

This command sequence creates two listing files called module1.lst and module2.lst; no object code is produced. These listing files are similar to normal listing files; however, the addresses shown are absolute addresses.

The absolute listing files created are module1.lst (see Example 10-1) and module2.lst (see Example 10-2).

Example 10-1. module1.lst

```
module1.abs                                                     PAGE    1

      21 002000             .text
      22                    .copy       "module1.asm"
 A     1 004000             .bss   array, 100
 A     2 004064             .bss   dflag, 2
 A     3                    .copy  globals.def
 B     1                    .global dflag
 B     2                    .global array
 B     3                    .global offset
 A     4 002000              .text

 A     5 002000 7640        MOV #offset,AC0
           002002 6608!
 A     6 002004 A000%       MOV dflag,AC0

 No Errors, No Warnings
```

Example 10-2. module2.lst

```
module2.abs                                                     PAGE    1

      21 002006             .text
      22                    .copy       "module2.asm"
 A     1 004066             .bss   offset, 2
 A     2                    .copy  globals.def
 B     1                    .global dflag
 B     2                    .global array
 B     3                    .global offset

 A     3 002006              .text
 A     4 002006 7640        MOV #offset,AC0
           002008 6680-
 A     5 00200a 7640        MOV #array,AC0
           00200c 0080!

 No Errors, No Warnings
```

Cross-Reference Lister Description

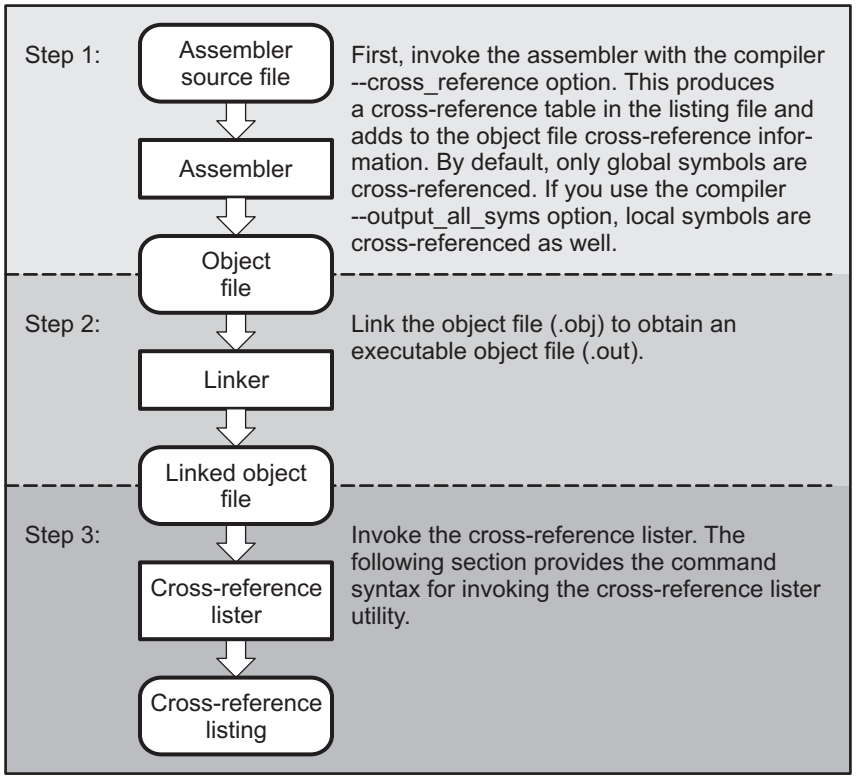
The TMS320C55x cross-reference lister is a debugging tool. This utility accepts linked object files as input and produces a cross-reference listing as output. This listing shows symbols, their definitions, and their references in the linked source files.

Topic **Page**

11.1 Producing a Cross-Reference Listing

Figure 11-1 illustrates the steps required to produce a cross-reference listing.

Figure 11-1. The 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 --cross_reference option. This option creates a cross-reference listing and adds cross-reference information to the object file. By default, the assembler cross-references only global symbols, but if the assembler is invoked with the --output_all_syms option, local symbols are also added. Link the object files to obtain an executable file.

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

xref55 [*options*] [*input filename* [*output filename*]]

xref55 is the command that invokes the cross-reference utility.

options identifies the cross-reference lister options you want to use. Options are not case sensitive and can appear anywhere on the command line following the command.

- **-l** (lowercase L) specifies the number of lines per page for the output file. The format of the -l option is -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** suppresses the banner and all progress information (run quiet).

input filename is a linked object file. If you omit the input filename, the utility prompts for a filename.

output filename is the name of the cross-reference listing file. If you omit the output filename, the default filename is the input filename with an .xrf extension.

11.3 Cross-Reference Listing Example

Example 11-1 is an example of cross-reference listing.

Example 11-1. Cross-Reference Listing

```
================================================================================

Symbol: INIT

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
file1.asm        EDEF   '000000    000080      3       1
file2.asm        EREF    000000    000080              2         11
================================================================================

Symbol: X

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
file1.asm        EREF    000000     000001              2         5
file2.asm        EDEF    000001     000001      5       1
================================================================================

Symbol: Y

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
file2.asm        EDEF    -000000     000080      7       1

================================================================================

Symbol: Z

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____   ________   ________    ______  _______  _______  _______
file2.asm        EDEF    000003     000003      9       1

================================================================================
```

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

Symbol Name of the symbol listed

Filename Name of the file where the symbol appears

RTYP The symbol's reference type in this file. The possible reference types are:

- **STAT** The symbol is defined in this file and is not declared as global.
- **EDEF** The symbol is defined in this file and is declared as global.
- **EREF** The symbol is not defined in this file but is referenced as global.
- **UNDF** The symbol is not defined in this file and is not declared as global.

AsmVal 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 can modify the contents of the object. A blank in this column indicates that the symbol was never used.

Table 11-1. Symbol Attributes in Cross-Reference Listing

Character	Meaning
'	Symbol defined in a .text section
"	Symbol defined in a .data section
+	Symbol defined in a .sect section
-	Symbol defined in a .bss or .usect section

Object File Utilities

This chapter describes how to invoke the following utilities:

- The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both text and XML formats.
- The **disassembler** accepts object files and executable files as input and produces an assembly listing as output. This listing shows assembly instructions, their opcodes, and the section program counter values.
- The **name utility** prints a list of names defined and referenced in an object file, executable files, and/or archive libraries.
- The **strip utility** removes symbol table and debugging information from object and executable files.

Topic **Page**

12.1 Invoking the Object File Display Utility

The object file display utility, *ofd55*, prints the contents of object files (.obj), executable files (.out), and/or archive libraries (.lib) in both text and XML formats. Hidden symbols are listed as *no name*, while localized symbols are listed like any other local symbol.

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

ofd55 [*options*] *input filename* [*input filename*]

ofd55 is the command that invokes the object file display utility.

input filename names the object file (.obj), executable file (.out), or archive library (.lib) source file. The filename must contain an extension.

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

--dwarf_display=*attributes* controls the DWARF display filter settings by specifying a comma-delimited list of *attributes*. When prefixed with no, an attribute is disabled instead of enabled.

Examples:
--dwarf_display=nodabbrev,nodline
--dwarf_display=all,nodabbrev
--dwarf_display=none,dinfo,types

The ordering of attributes is important (see --obj_display). The list of available display attributes can be obtained by invoking ofd55 --dwarf_display=help.

-g appends DWARF debug information to program output.

-h displays help

-o=*filename* sends program output to *filename* rather than to the screen.

--obj_display *attributes* controls the object file display filter settings by specifying a comma-delimited list of *attributes*. When prefixed with no, an attribute is disabled instead of enabled.

Examples:
--obj_display=rawdata,nostrings
--obj_display=all,norawdata
--obj_display=none,header

The ordering of attributes is important. For instance, in "--obj_display=none,header", ofd55 disables all output, then re-enables file header information. If the attributes are specified in the reverse order, (header,none), the file header is enabled, the all output is disabled, including the file header. Thus, nothing is printed to the screen for the given files. The list of available display attributes can be obtained by invoking ofd55 --obj_display=help.

-v prints verbose text output.

-x displays output in XML format.

--xml_indent=*num* sets the number of spaces to indent nested XML tags.

If an archive file is given as input to the object file display utility, each object file member of the archive is processed as if it was passed on the command line. The object file members are processed in the order in which they appear in the archive file.

If the object file display utility is invoked without any options, it displays information about the contents of the input files on the console screen.

Object File Display Format

NOTE: The object file display utility produces data in a text format by default. This data is not intended to be used as input to programs for further processing of the information. XML format should be used for mechanical processing.

12.2 Invoking the Disassembler

The disassembler, *dis55*, examines the output of the assembler or linker. This utility accepts an object file or executable file as input and writes the disassembled object code to standard output or a specified file.

To invoke the disassembler, enter the following:

dis55 [*options*] *input filename*[.] [*output filename*]

dis55 is the command that invokes the disassembler.

options identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The name utility options are as follows:

- **-a** enables the printing of branch destination addresses along with labels within instructions.
- **-b** displays data as bytes instead of words.
- **-c** includes a COFF file description at the top of the listing. This description includes information on the memory model, relocation, line numbers, and local symbols.
- **-d** disables display of data sections.
- **-e** displays integer values in hexadecimal.
- **-g** displays in algebraic assembler format.
- **-h** shows the current help screen.
- **-i** disassembles .data sections as instructions.
- **-I** disassembles code sections as data.
- **-l** displays load rather than run address.
- **-L** displays both load and run addresses.
- **-m** displays in mnemonic assembler format (default).
- **-n** dumps the symbol table.
- **-q** (quiet mode) suppresses the banner and all progress information.
- **-qq** (super quiet mode) suppresses all headers.
- **-r** uses the compiler's convention of enabling the ARMS and CPL bits, and disabling the C54CM bit. By default, the disassembler assumes the status bits have their hardware reset values; that is, ARMS and CPL are off and C54CM is on. Use -r when disassembling any file generated from C/C++ source.
- **-s** suppresses printing of opcode and section program counter in the listing. When you use this option along with –qq, the disassembly listing looks like the original assembly source file.
- **-t** suppresses the display of text sections in the listing.

input filename[.*ext*] is the name of the input file. If the optional extension is not specified, the file is searched for in this order:

1. *infile*
2. *infile*.out, an executable file
3. *infile*.obj, an object file

output filename is the name of the optional output file to which the disassembly will be written. If an output filename is not specified, the disassembly is written to standard output.

Consider the following assembly source file called test.asm:

```
          .global GLOBAL
          .global FUNC
CONSTANT  .set   1
          .text
START     MOV    AR1,AR0
          ADD    #CONSTANT,AC0
last      ADD    #GLOBAL,AC0
          .data
          .word  4
foo       .word  1
          .word  FUNC
```

The symbols GLOBAL and FUNC are defined in test2.asm:

```
         .global GLOBAL
         .global FUNC
GLOBAL   .set   100
FUNC:    RETURN
```

The examples below assume that test.asm and test2.asm have been assembled and linked with the following commands:

```
cl55 --quiet --output_all_syms test.asm
cl55 --quiet --output_all_syms test2.asm
cl55 --run_linker --quiet test.obj test2.obj --output_file=test.out
```

Example 1

To create a standard disassembly listing of an object file, enter:

```
dis55 test.obj
```

In the result below, the value 1 was encoded into the first ADD instruction, and the 16-bit ADD instruction was used. For the second ADD instruction, the use of the global symbol GLOBAL caused the assembler to use the 32-bit ADD instruction. The symbols GLOBAL and FUNC are resolved by the linker.

```
TMS320C55x COFF Disassembler              Version x.xx
Copyright (c) 1996-2007 Texas Instruments Incorporated
Disassembly of test.obj:

TEXT Section .text, 0x8 bytes at 0x0
000000:          START:
000000: 2298        MOV AR1,AR0
000002: 4010        ADD #1,AC0
000004:          last:
000004: 7b000000    ADD #0,AC0,AC0
DATA Section .data, 0x3 words at 0x0
000000: 0004        .word 0x0004
000001:          foo:
000001: 0001        .word 0x0001
000002: 0000        .word 0x0000
```

Example 2

You can view the object file information with the -c option. The -q option suppresses the printing of the banner.

```
dis55 -qc test.obj
```

```
>> Target is C55x Phase 3, mem=small, call=c55_std
   Relocation information may exist in file
   File is not executable
   Line number information may be present in the file
   Local symbols may be present in the file
```

Example 3

To create a standard disassembly listing of an executable file, enter:

```
dis55 -q test.out
```

```
TEXT Section .text, 0x8 bytes at 0x0
000000:          START:
000000: 2298        MOV AR1,AR0
000002: 4010        ADD #1,AC0
000004:          last:
000004: 7b000000    ADD #0,AC0,AC0

DATA Section .data, 0x3 words at 0x0
000000: 0004        .word 0x0004
000001:          foo:
000001: 0001        .word 0x0001
000002: 0000        .word 0x0000
```

```
TEXT Section .text, 0xB bytes at 0x100 000100: START: 000100: 2298 MOV AR1,AR0 000102: 4010 ADD
#1,AC0 000104: last: 000104: 7b006400 ADD #100,AC0,AC0 000108: FUNC: 000108: 4804 RET 00010a: 20
NOP 00010b: ___etext__: 00010b: etext: DATA Section .data, 0x3 words at 0x8000 008000: 0004 .word
0x0004 008001: foo: 008001: 0001 .word 0x0001 008002: 0108 .word 0x0108
```

In the above code, the disassembly listing displays the addresses used by the instructions and data, as well as the resolved symbol values in the ADD instruction and in the final .word directive. The .word directive contains the correct address of the function. The NOP in the .text section is used to pad the section.

12.3 Invoking the Name Utility

The name utility, *nm55*, prints the list of names defined and referenced in an object file, executable file, or archive library. It also prints the symbol value and an indication of the kind of symbol. Hidden symbols are listed as "".

To invoke the name utility, enter the following:

nm55 [*-options*] [*input filenames*]

nm55 is the command that invokes the name utility.

input filename is an object file (.obj), executable file (.out), or archive library (.lib).

options identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The name utility options are as follows:

- **-a** prints all symbols.
- **-c** also prints C_NULL symbols for a COFF object module.
- **-d** also prints debug symbols for a COFF object module.
- **-f** prepends file name to each symbol.
- **-g** prints only global symbols.
- **-h** shows the current help screen.
- **-l** produces a detailed listing of the symbol information.
- **-n** sorts symbols numerically rather than alphabetically.
- **-o** *file* outputs to the given file.
- **-p** causes the name utility to not sort any symbols.
- **-q** (quiet mode) suppresses the banner and all progress information.
- **-r** sorts symbols in reverse order.
- **-t** also prints tag information symbols for a COFF object module.
- **-u** only prints undefined symbols.

12.4 Invoking the Strip Utility

The strip utility, *strip55*, removes symbol table and debugging information from object and executable files.

To invoke the strip utility, enter the following:

strip55 [**-p**] *input filename* [*input filename*]

strip55 is the command that invokes the strip utility.

input filename is an object file (.obj) or an executable file (.out).

options identifies the strip utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The strip utility option is as follows:

- **-o** *filename* writes the stripped output to filename.
- **-p** removes all information not required for execution. This option causes more information to be removed than the default behavior, but the object file is left in a state that cannot be linked. This option should be used only with executable (.out) files.

When the strip utility is invoked without the -o option, the input object files are replaced with the stripped version.

Hex Conversion Utility Description

The TMS320C55x assembler and linker create object files which are in binary formats that encourage modular programming and provide powerful and flexible methods for managing code segments and target system memory.

Most EPROM programmers do not accept object files as input. The hex conversion utility converts an 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 an object file (for example, when using debuggers and loaders).

The hex conversion utility can produce these output file formats:

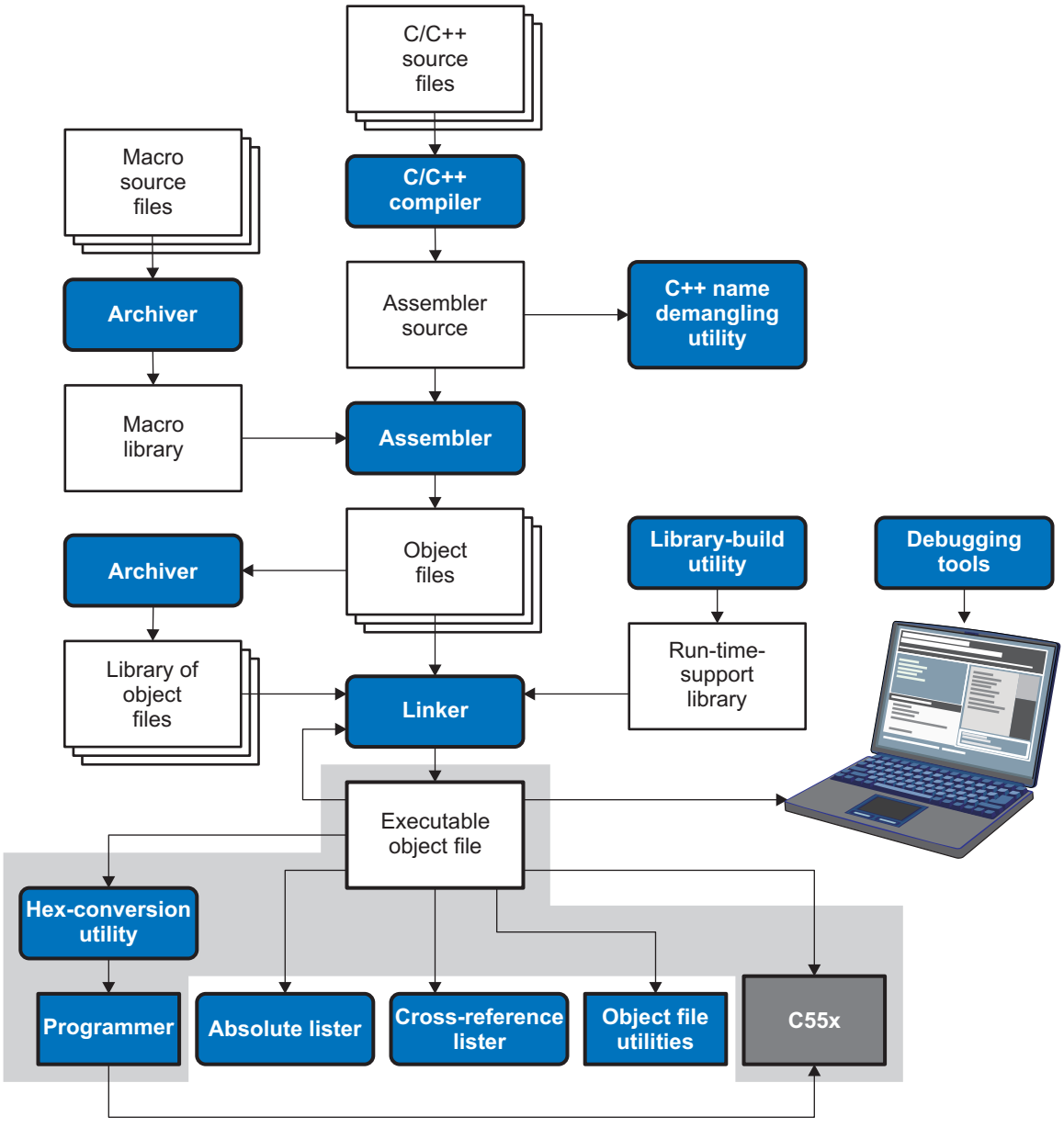
- ASCII-Hex, supporting 16-bit addresses
- Extended Tektronix (Tektronix)
- Intel MCS-86 (Intel)
- Motorola Exorciser (Motorola-S), supporting 16-bit, 24-bit, and 32-bit addresses
- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses
- Texas Instruments TI-TXT format, supporting 16-bit addresses

Topic **Page**

13.1 The Hex Conversion Utility's Role in the Software Development Flow

Figure 13-1 highlights the role of the hex conversion utility in the software development process.

Figure 13-1. The Hex Conversion Utility in the TMS320C55x Software Development Flow



13.2 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.

```
hex55 -t firmware -o firm.lsb -o firm.msb
```

- **Specify the options and filenames in a command file.** You can create a 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:

```
hex55 hexutil.cmd
```

In addition to regular command line information, you can use the hex conversion utility ROMS and SECTIONS directives in a command file.

13.2.1 Invoking the Hex Conversion Utility From the Command Line

To invoke the hex conversion utility, enter:

hex55 [*options*] *filename*

hex55 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. Table 13-1 lists the basic options.

- All options are preceded by a hyphen 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 multi-character 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 --quiet option, which must be used before any other options.

filename names an object file or a command file (for more information, see Section 13.2.2).

Table 13-1. Basic Hex Conversion Utility Options

Option	Alias	Description	See
		General Options	
--entry_point=*addr*	-e	Specify entry point address or symbol name	Section 13.10.3
--exclude={*fname*(*sname*) | *sname*}	-exclude	If the filename (*fname*) is omitted, all sections matching *sname* will be excluded.	Section 13.7
--fill=*value*	-fill	Fill holes with *value*	Section 13.9.2
--help	-options, -h	Display the syntax for invoking the utility and list available options. If the option is followed by another option or phrase, detailed information about that option or phrase is displayed. For example, to see information about options associated with generating a boot table, use --help boot.	Section 13.2.2
--image	-image	Select image mode	Section 13.9.1
--linkerfill	-linkerfill	Include linker fill sections in images	--
--map=*filename*	-map	Generate a map file	Section 13.4.2
--memwidth=*value*	-memwidth	Define the system memory word width (default 16 bits)	Section 13.3.3
--olength=*value*	-olength	Specify maximum number of data items per line of output	--
--order={LS|MS}	-order	Specify data ordering (endianness)	Section 13.3.6
--outfile=*filename*	-o	Specify an output filename	Section 13.8
--quiet	-q	Run quietly (when used, it must appear *before* other options)	Section 13.2.2

Table 13-1. Basic Hex Conversion Utility Options (continued)

Option	Alias	Description	See
--romwidth=*value*	-romwidth	Specify the ROM device width (default depends on format used)	Section 13.3.4
--zero	-zero, -z	Reset the address origin to 0 in image mode	Section 13.9.3
		Diagnostic Options	
--diag_error=*id*		Categorizes the diagnostic identified by *id* as an error	Section 13.12
--diag_remark=*id*		Categorizes the diagnostic identified by *id* as a remark	Section 13.12
--diag_suppress=*id*		Suppresses the diagnostic identified by *id*	Section 13.12
--diag_warning=*id*		Categorizes the diagnostic identified by *id* as a warning	Section 13.12
--display_error_number		Displays a diagnostic's identifiers along with its text	Section 13.12
--issue_remarks		Issues remarks (nonserious warnings)	Section 13.12
--no_warnings		Suppresses warning diagnostics (errors are still issued)	Section 13.12
--set_error_limit=*count*		Sets the error limit to *count*. The linker abandons linking after this number of errors. (The default is 100.)	Section 13.12
		Boot Table Options	
--boot	-boot	Convert all sections into bootable form (use instead of a SECTIONS directive)	Section 13.10.3
--bootorg=*addr*	-bootorg	Specify origin address or symbol of boot table in ROM	Section 13.10.3
--bootpage	-bootpage	Specify page of boot table in ROM	Section 13.10.3
--delay *value*	-delay	Specify delay of given *value* in configuration table	Section 13.10.3
-parallel16		Specify a 16-bit parallel interface boot table (-memwidth 16 and -romwidth 16)	Section 13.10.3
-parallel32		Specify a 32-bit parallel interface boot table (-memwidth 16 and -romwidth 32)	Section 13.10.3
-reg_config *addr*,*data*{,*data*,...}		Initialize locations starting at location addr with the given data values	Section 13.10.3
--serial8	-serial8	Specify an 8-bit serial interface boot table (-memwidth 8 and -romwidth 8)	Section 13.10.3
--serial16	-serial16	Specify an 16-bit serial interface boot table (-memwidth 16and -romwidth 16)	Section 13.10.3
--version*device*	-v	Specify the silicon boot table version	Section 13.10.3
		Output Options	
--ascii	-a	Select ASCII-Hex	Section 13.13.1
--binary	-b	Select binary (Must have memory width of 8 bits.)	--
--intel	-i	Select Intel	Section 13.13.2
--motorola=1	-m1	Select Motorola-S1	Section 13.13.3
--motorola=2	-m2	Select Motorola-S2 (default -m option)	Section 13.13.3
--motorola=3	-m3	Select Motorola-S3	Section 13.13.3
--tektronix	-x	Select Tektronix (default format when no output option is specified)	Section 13.13.4
--ti_tagged	-t	Select TI-Tagged	Section 13.13.5
--ti_txt		Select TI-Txt	Section 13.13.6
		Load Image Options	
--load_image		Select load image	Section 13.6
--section_name_prefix=*string*		Specify the section name prefix for load image object files	Section 13.6

13.2.2 Invoking the Hex Conversion Utility With a Command File

A command file is useful if you plan to invoke the utility more than once with the same input files and options. It is also useful if you want to use the ROMS and SECTIONS hex conversion utility directives to customize the conversion process.

Command files are ASCII files that contain one or more of the following:

- **Options and filenames.** These are specified in a command file in exactly the same manner as on the command line.
- **ROMS directive.** The ROMS directive defines the physical memory configuration of your system as a list of address-range parameters. (See Section 13.4.)

  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 13.10.)
- **SECTIONS directive.** The hex conversion utility SECTIONS directive specifies which sections from the object file are selected. (See Section 13.5.)
- **Comments.** You can add comments to your command file by using the /* and */ delimiters. For example:

```
/*   This is a comment.   */
```

To invoke the utility and use the options you defined in a command file, enter:

hex55 *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:

```
hex55 firmware.cmd --map=firmware.mxp
```

The order in which these options and filenames appear is not important. The utility reads all input from the command line and all information from the command file before starting the conversion process. However, if you are using the -q option, *it must appear as the first option on the command line or in a command file.*

The **--help** option displays the syntax for invoking the compiler and lists available options. If the --help option is followed by another option or phrase, detailed information about the option or phrase is displayed. For example, to see information about options associated with generating a boot table use --help boot.

The **--quiet** option suppresses the hex conversion utility's normal banner and progress information.

- Assume that a command file named firmware.cmd contains these lines:

```
firmware.out         /* input file  */
--ti-tagged          /* TI-Tagged   */
--outfile=firm.lsb   /* output file 1, LSBs of ROM */
--outfile=firm.msb   /* output file 2, MSBs of ROM */
```

  You can invoke the hex conversion utility by entering:

```
hex55 firmware.cmd
```

- This example shows how to convert 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   */
--intel           /* Intel format */
--map=appl.mxp    /* map file     */

ROMS
{
   ROW1: origin=0x1000 len=0x4000 romwidth=8
         files={ appl.u0 appl.u1 }
   ROW2: origin=0x5000 len=0x4000 romwidth=8
         files={ app1.u2 appl.u3 }
}

SECTIONS
{   .text: BOOT
    .data, .cinit, .sect1, .vectors, .const:
}
```

13.3 Understanding Memory Widths

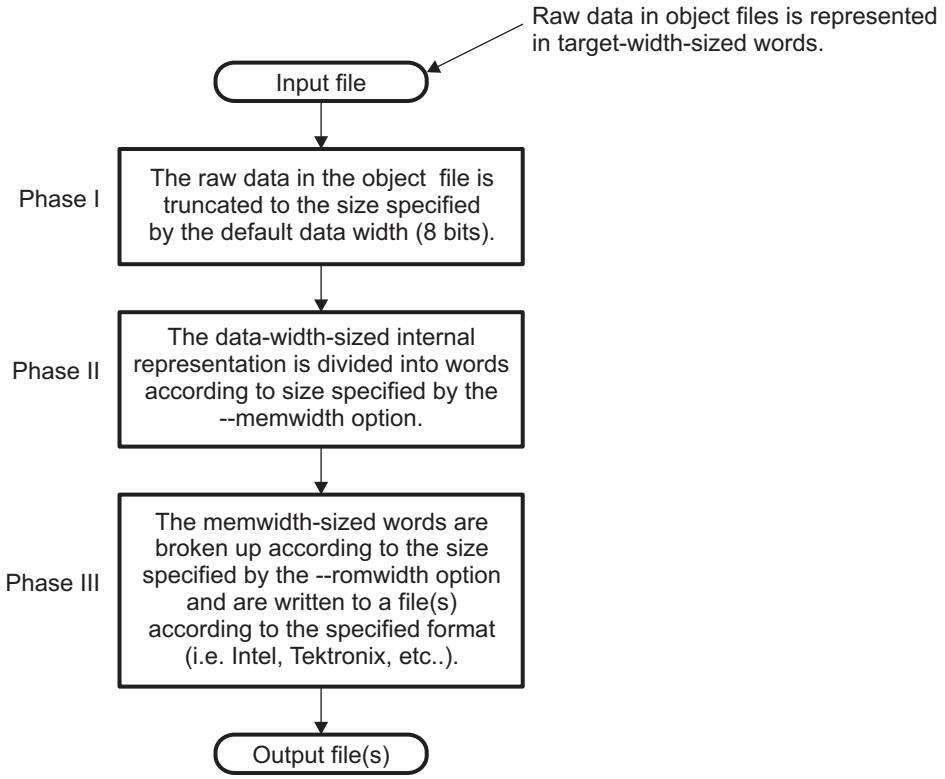
The hex conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex conversion utility, you must understand how the utility treats word widths. Three widths are important in the conversion process:

- Target width
- Data width
- Memory width
- ROM width

The terms target word, data word, memory word, and ROM word refer to a word of such a width.

Figure 13-2 illustrates the separate and distinct phases of the hex conversion utility's process flow.

Figure 13-2. Hex Conversion Utility Process Flow



13.3.1 Target Width

Target width is the unit size (in bits) of the target processor's word. The unit size corresponds to the data bus size on the target processor. The width is fixed for each target and cannot be changed. The TMS320C55x targets have a width of 16 bits.

13.3.2 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 8 bits for the TMS320C55x and cannot be changed. A data width of 8 is used to support processing of information in COFF code sections, which consist of 8-bit bytes.

13.3.3 Specifying the Memory Width

Memory width is the physical width (in bits) of the memory system. Usually, the memory system is physically the same width as the target processor width: a 16-bit processor has a 16-bit memory architecture. However, some applications require target words to be broken into multiple, consecutive, and narrower memory words. The C55x has a target width of 16 because the data memory is addressed as 16-bit words. However, the memory itself can also be addressed as 8-bit bytes and sometimes the memory width may need to be narrower than the target width.

By default, the hex conversion utility sets memory width to the target width (in this case, 16 bits).

You can change the memory width (except for TI-TXT format) 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 13.4.

For both methods, use a value that is a power of 2 greater than or equal to 8.

You should change the memory width default value of 16 only in special situations when you need to break single target words into consecutive, narrower memory words. The most common situation in which memory words are narrower than target words is when you use an on-chip boot loader that supports booting from narrower memory. For example, a 16-bit TMS320C55x 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).

Binary Format is 8 Bits Wide

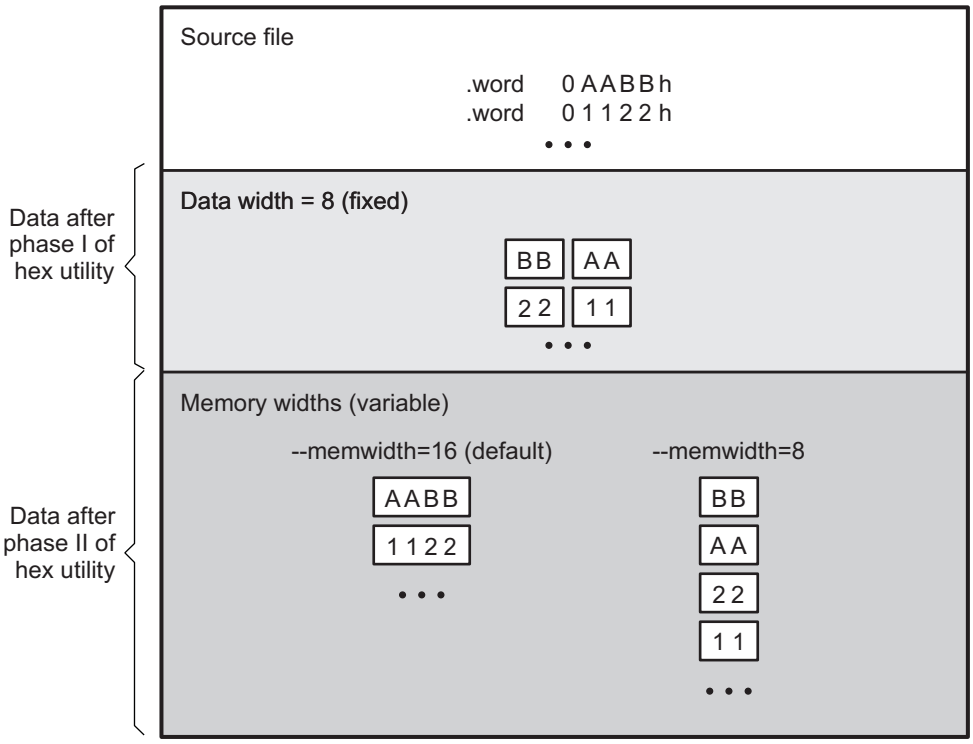
NOTE: You cannot change the memory width of the Binary format. The Binary hex format supports an 8-bit memory width only.

TI-TXT Format is 8 Bits Wide

NOTE: You cannot change the memory width of the TI-TXT format. The TI-TXT hex format supports an 8-bit memory width only.

Figure 13-3 demonstrates how the memory width is related to object file data.

Figure 13-3. Object File Data and Memory Widths



13.3.4 Partitioning Data Into Output Files

ROM width specifies the physical width (in bits) of each ROM device and corresponding output file (usually one byte or eight bits). The ROM width determines how the hex conversion utility partitions the data into output files. After the object file data is mapped to the memory words, the memory words are broken into one or more output files. The number of output files is determined by the following formulas:

- If memory width ≥ ROM width:

  number of files = memory width ÷ ROM width
- If memory width < ROM width:

  number of files = 1

For example, for a memory width of 16, you could specify a ROM width value 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.

The TI-Tagged Format is 16 Bits Wide

NOTE: You cannot change the ROM width of the TI-Tagged format. The TI-Tagged format supports a 16-bit ROM width only.

TI-TXT Format is 8 Bits Wide

NOTE: You cannot change the ROM width of the TI-TXT format. The TI-TXT hex format supports only an 8-bit ROM width.

You can change ROM width (except for TI-Tagged and TI-TXT formats) by:

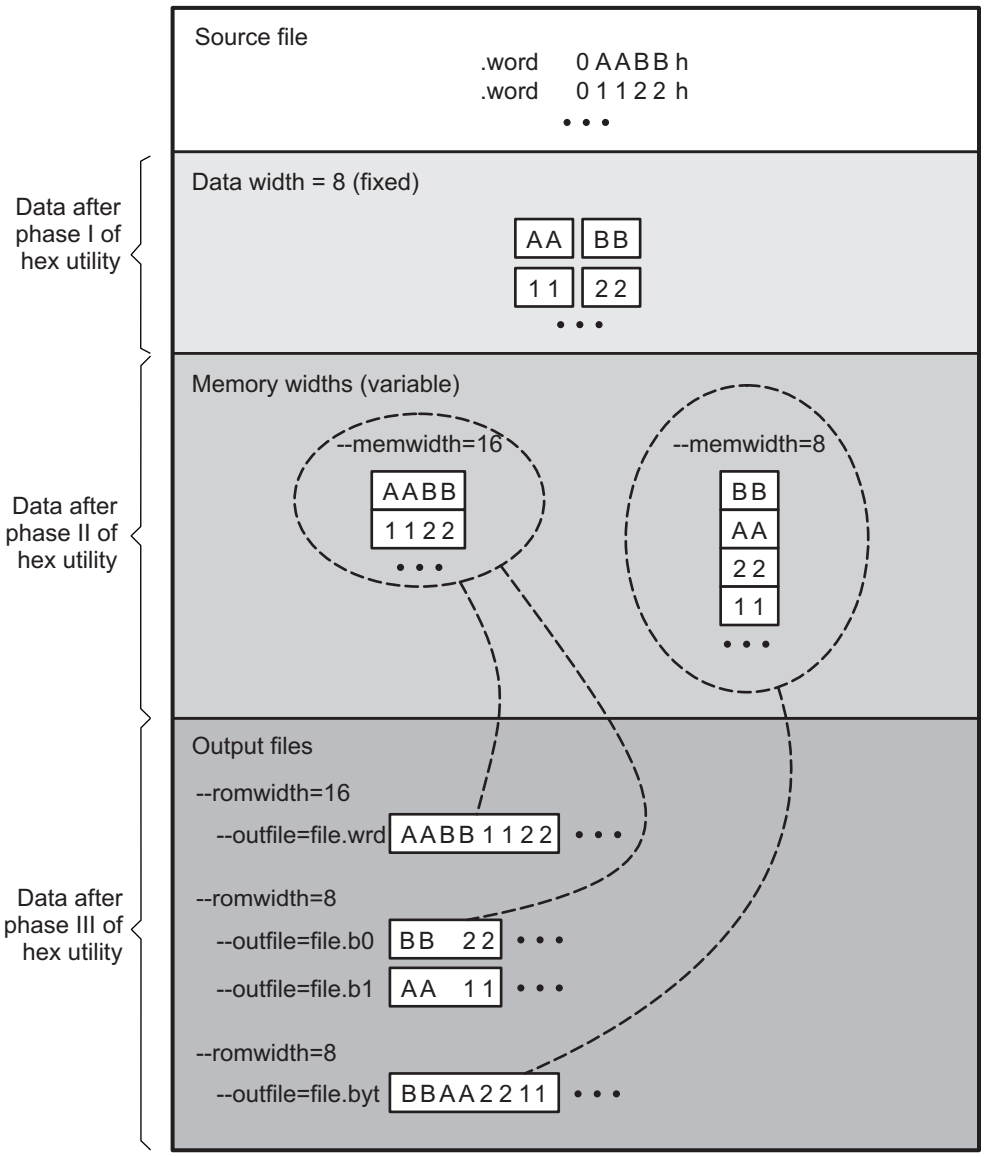
- Using the **--romwidth** option. This option changes the ROM width value for the entire object file.
- Setting the **romwidth** parameter of the ROMS directive. This parameter changes the ROM width value for a specific ROM address range and overrides the --romwidth option for that range. See Section 13.4.

For both methods, use a value that is a power of 2 greater than or equal to 8.

If you select a ROM width that is wider than the natural size of the output format (16 bits for TI-Tagged or 8 bits for all others), the utility simply writes multibyte fields into the file.

Figure 13-4 illustrates how the object file data, memory, and ROM widths are related to one another.

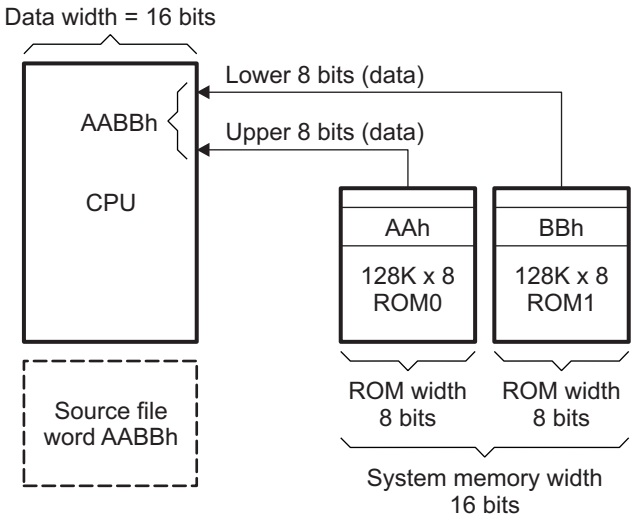
Figure 13-4. Data, Memory, and ROM Widths



13.3.5 A Memory Configuration Example

Figure 13-5 shows a typical memory configuration example. This memory system consists of two 128K × 8-bit ROM devices.

Figure 13-5. C55x Memory Configuration Example



13.3.6 Specifying Word Order for Output Words

There are two ways to split a wide word into consecutive memory locations in the same hex conversion utility output file:

- **--order=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 least significant part of the wide word occupies the first of the consecutive locations.

By default, the utility uses big-endian format. Unless your boot loader program expects little-endian format, avoid using --order=LS.

NOTE: **When the --order Option Applies**

- This option applies only when you use a memory width with a value.
- 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.

13.4 The ROMS Directive

The ROMS directive specifies the physical memory configuration of your system as a list of address-range parameters.

Each address range produces one set of files containing the hex conversion utility output data that corresponds to that address range. Each file can be used to program one single ROM device.

The ROMS directive is similar to the MEMORY directive of the TMS320C55x 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**=**{** *filename* $_1$, *filename* $_2$, ...**}**]
romname **:** [**origin**=*value*,] [**length**=*value*,] [**romwidth**=*value*,]
[**memwidth**=*value*,] [**fill**=*value*]
[**files**=**{** *filename* $_1$, *filename* $_2$, ...**}**]
...
}

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 can be one to eight characters in length. The name has no significance to the program; it simply identifies the range, except when the output is for a load image in which case it denotes the section name. (Duplicate memory range names are allowed.)

origin specifies the starting address of a memory range. It can be entered as origin, org, or o. The associated value must be a decimal, octal, or hexadecimal constant. If you omit the origin value, the origin defaults to 0. The following table summarizes the notation you can use to specify a decimal, octal, or hexadecimal constant:

Constant	Notation	Example
Hexadecimal	0x prefix or h suffix	0x77 or 077h
Octal	0 prefix	077
Decimal	No prefix or suffix	77

length specifies the length of a memory range as the physical length of the ROM device. It can be entered as length, len, or l. The value must be a decimal, octal, or hexadecimal constant. If you omit the length value, it defaults to the length of the entire address space.

romwidth specifies the physical ROM width of the range in bits (see Section 13.3.4). 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 Section 13.3.3). 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.* (See Section 13.5.)

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. A hole is an area between the input sections that comprises an output section that contains no actual code or data. The fill value must be a decimal, octal, or hexadecimal constant with a width equal to the target width. Any value you specify here overrides the --fill option. When using fill, you must also use the --image command line option. (See Section 13.9.2.)

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, where the bits of the memory word are numbered from right to left. The number of file names must equal the number of output files that the range generates. To calculate the number of output files, see Section 13.3.4. The utility warns you if you list too many or too few filenames.

Unless you are using the --image option, all of the parameters that define a range are optional; the commas and equal signs are also optional. A range with no origin or length defines the entire address space. In image mode, an origin and length are required for all ranges.

Ranges must not overlap and must be listed in order of ascending address.

13.4.1 When to Use the ROMS Directive

If you do not use a ROMS directive, the utility defines a single default range that includes the entire address space. This is equivalent to a ROMS directive with a single range without origin or length.

Use the ROMS directive when you want to:

- **Program large amounts of data into fixed-size ROMs**. When you specify memory ranges corresponding to the length of your ROMs, the utility automatically breaks the output into blocks that fit into the ROMs.
- **Restrict output to certain segments**. You can also use the ROMS directive to restrict the conversion to a certain segment or segments of the target address space. The utility does not convert the data that falls outside of the ranges defined by the ROMS directive. Sections can span range boundaries; the utility splits them at the boundary into multiple ranges. If a section falls completely outside any of the ranges you define, the utility does not convert that section and issues no messages or warnings. Thus, you can exclude sections without listing them by name with the SECTIONS directive. However, if a section falls partially in a range and partially in unconfigured memory, the utility issues a warning and converts only the part within the range.
- **Use image mode.** When you use the --image option, you must use a ROMS directive. Each range is filled completely so that each output file in a range contains data for the whole range. Holes before, between, or after sections are filled with the fill value from the ROMS directive, with the value specified with the --fill option, or with the default value of 0.

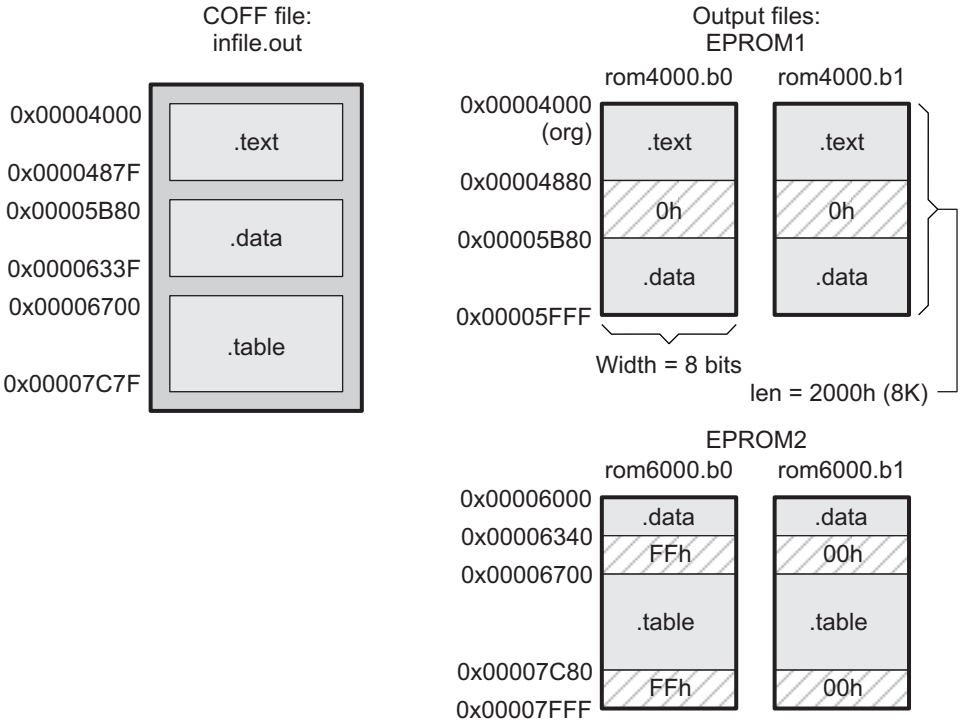
13.4.2 An Example of the ROMS Directive

The ROMS directive in Example 13-1 shows how 16K bytes of 16-bit memory could be partitioned for two 8K-byte 8-bit EPROMs. Figure 13-6 illustrates the input and output files.

Example 13-1. A ROMS Directive Example

```
infile.out
--image
--memwidth 16

ROMS
{
   EPROM1: org = 0x4000, len = 0x2000, romwidth = 8
           files = { rom4000.b0, rom4000.b1}

   EPROM2: org = 0x6000, len = 0x2000, romwidth = 8,
           fill = 0xFF00,
           files = { rom6000.b0, rom6000.b1}
}
```

Figure 13-6. The infile.out File Partitioned Into Four Output Files



The map file (specified with the --map option) is advantageous when you use the ROMS directive with multiple ranges. The map file shows each range, its parameters, names of associated output files, and a list of contents (section names and fill values) broken down by address. Example 13-2 is a segment of the map file resulting from the example in Example 13-1.

Example 13-2. Map File Output From Example 13-1 Showing Memory Ranges

```
-------------------------------------------------------
00004000..00005fff Page=0 Width=8 "EPROM1"
-------------------------------------------------------
   OUTPUT FILES:   rom4000.b0   [b0..b7]
                   rom4000.b1   [b8..b15]
   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
-------------------------------------------------------
00006000..00007fff Page=0 Width=8 "EPROM2"
-------------------------------------------------------
   OUTPUT FILES:   rom6000.b0   [b0..b7]
                   rom6000.b1   [b8..b15]
   CONTENTS: 00006000..0000633f .data
             00006340..000066ff FILL = ff00ff00
             00006700..00007c7f .table
             00007c80..00007fff FILL = ff00ff00
```

EPROM1 defines the address range from 0x00004000 through 0x00005FFF with the following sections:

This section ...	Has this range ...
.text	0x00004000 through 0x0000487F
.data	0x00005B80 through 0x00005FFF

The rest of the range is filled with 0h (the default fill value), converted into two output files:

- rom4000.b0 contains bits 0 through 7
- rom4000.b1 contains bits 8 through 15

EPROM2 defines the address range from 0x00006000 through 0x00007FFF with the following sections:

This section ...	Has this range ...
.data	0x00006000 through 0x0000633F
.table	0x00006700 through 0x00007C7F

The rest of the range is filled with 0xFF0 (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

13.5 The SECTIONS Directive

You can convert specific sections of the object file by name with the hex conversion utility SECTIONS directive. You can also specify those sections that you want to locate in ROM at a different address than the *load* address specified in the linker command file. If you:

- Use a SECTIONS directive, the utility converts only the sections that you list in the directive and ignores all other sections in the object file.
- Do not use a SECTIONS directive, the utility converts all initialized sections that fall within the configured memory.

Uninitialized sections are *never* converted, whether or not you specify them in a SECTIONS directive.

Sections Generated by the C/C++ Compiler

NOTE: The TMS320C55x C/C++ compiler automatically generates these sections:

- **Initialized sections:** .text, .const, .cinit, .pinit (C++ only), and .switch
- **Uninitialized sections:** .bss, .stack, .sysstack, and .sysmem

Use the SECTIONS directive in a command file. (See Section 13.2.2.) The general syntax for the SECTIONS directive is:

SECTIONS
{
oname(*sname*)[**:**] [**paddr**=*value*]
oname(*sname*)[**:**] [**paddr**= **boot**]
oname(*sname*)[**:**] [**boot**]
...
}

SECTIONS	begins the directive definition.
oname	identifies the object filename the section is located within. The filename is optional when only a single input file is given, but required otherwise.
sname	identifies a section in the input file. If you specify a section that does not exist, the utility issues a warning and ignores the name.
paddr=*value*	specifies the physical ROM address at which this section should be located. This value overrides the section load address given by the linker. 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 a 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 a boot loader. This is equivalent to using **paddr=boot**. Boot sections have a physical address determined by the location of the boot table. The origin of the boot table is specified with the --bootorg option.

For more similarity with the linker's SECTIONS directive, you can use colons after the section names (in place of the equal sign on the boot keyboard). For example, the following statements are equivalent:

```
SECTIONS { .text: .data: boot }
SECTIONS { .text: .data = boot }
```

In the example below, the object file contains six initialized sections: .text, .data, .const, .vectors, .coeff, and .tables. Suppose you want only .text and .data to be converted. Use a SECTIONS directive to specify this:

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

To configure both of these sections for boot loading, add the boot keyword:

```
SECTIONS { .text = boot .data = boot }
```

Using the --boot Option and the SECTIONS Directive

NOTE: When you use the SECTIONS directive with the boot table (--boot) option, the --boot option is ignored. You must explicitly specify any boot sections in the SECTIONS directive. For more information about --boot and other command line options associated with boot tables, see Section 13.2 and Section 13.10.

13.6 The Load Image Format (--load_image Option)

A load image is an object file which contains the load addresses and initialized sections of one or more executable files. The load image object file can be used for ROM masking or can be relinked in a subsequent link step.

13.6.1 Load Image Section Formation

The load image sections are formed by collecting the initialized sections from the input executables. There are two ways the load image sections are formed:

- **Using the ROMS Directive**. Each memory range that is given in the ROMS directive denotes a load image section. The romname is the section name. The origin and length parameters are required. The memwidth, romwidth, and files parameters are invalid and are ignored.

  When using the ROMS directive and the load_image option, the --image option is required.
- **Default Load Image Section Formation**. If no ROMS directive is given, the load image sections are formed by combining contiguous initialized sections in the input executables. Sections with gaps smaller than the target word size are considered contiguous.

  The default section names are image_1, image_2, ... If another prefix is desired, the --section_name_prefix=*prefix* option can be used.

13.6.2 Load Image Characteristics

All load image sections are initialized data sections. In the absence of a ROMS directive, the load/run address of the load image section is the load address of the first input section in the load image section. If the SECTIONS directive was used and a different load address was given using the paddr parameter, this address will be used.

The load image format always creates a single load image object file. The format of the load image object file is determined based on the input files. The file is not marked executable and does not contain an entry point. The default load image object file name is ti_load_image.obj. This can be changed using the --outfile option. Only one --outfile option is valid when creating a load image, all other occurrences are ignored.

Concerning Load Image Format

NOTE: These options are invalid when creating a load image:

- --memwidth
- --romwidth
- --order
- --zero
- --byte

If a boot table is being created, either using the SECTIONS directive or the --boot option, the ROMS directive must be used.

13.7 Excluding a Specified Section

The --exclude *section_name* option can be used to inform the hex utility to ignore the specified section. If a SECTIONS directive is used, it overrides the --exclude option.

For example, if a SECTIONS directive containing the section name *mysect* is used and an --exclude *mysect* is specified, the SECTIONS directive takes precedence and *mysect* is not excluded.

The --exclude option has a limited wildcard capability. The * character can be placed at the beginning or end of the name specifier to indicate a suffix or prefix, respectively. For example, --exclude sect* disqualifies all sections that begin with the characters sect.

If you specify the --exclude option on the command line with the * wildcard, enter quotes around the section name and wildcard. For example, --exclude"sect*". Using quotes prevents the * from being interpreted by the hex conversion utility. If --exclude is in a command file, then the quotes should not be specified.

If multiple object files are given, the object file in which the section to be excluded can be given in the form oname(sname). If the object filename is not provided, all sections matching the section name are excluded. Wildcards cannot be used for the filename, but can appear within the parentheses.

13.8 Assigning Output Filenames

When the hex conversion utility translates your object file into a data format, it partitions the data into one or more output files. When multiple files are formed by splitting memory words into ROM words, *filenames are always assigned in order from least to most significant*, where bits in the memory words are numbered from right to left. This is true, regardless of target or endian ordering.

The hex conversion utility follows this sequence when assigning output filenames:

1. **It looks for the ROMS directive.** If a file is associated with a range in the ROMS directive and you have included a list of files (files = {. . .}) on that range, the utility takes the filename from the list.

   For example, assume that the target data is 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 to xyz.b0 and the most significant bits to xyz.b1.

2. **It looks for the --outfile options.** You can specify names for the output files by using the --outfile option. If no filenames are listed in the ROMS directive and you use --outfile options, the utility takes the filename from the list of --outfile options. The following line has the same effect as the example above using the ROMS directive:

   ```
   --outfile=xyz.b0 --outfile=xyz.b1
   ```

   If both the ROMS directive and --outfile options are used together, the ROMS directive overrides the --outfile 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 input file plus a 2- to 3-character extension. The extension has three parts:

   (a) A format character, based on the output format (see Section 13.13):

a	for ASCII-Hex
i	for Intel
m	for Motorola-S
t	for TI-Tagged
x	for Tektronix

(b) The range number in the ROMS directive. Ranges are numbered starting with 0. If there is no ROMS directive, or only one range, the utility omits this character.

(c) The file number in the set of files for the range, starting with 0 for the least significant file.

For example, assume a.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 a.i0, a.i1, a.i2, a.i3.

If you include the following ROMS directive when you invoke the hex conversion utility, you would have four output files:

```
ROMS
{
   range1: o = 0x1000 l = 0x1000
   range2: o = 0x2000 l = 0x1000
}
```

These output files ...	Contain data in these locations ...
a.i00 and a.i01	0x1000 through 0x1FFF
a.i10 and a.i11	0x2000 through 0x2FFF

13.9 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.

13.9.1 Generating a Memory Image

With the --image option, the utility generates a memory image by completely filling all of the mapped ranges specified in the ROMS directive.

An object file consists of blocks of memory (sections) with assigned memory locations. Typically, all sections are not adjacent: there are holes between sections in the address space for which there is no data. When such a file is converted *without* the use of image mode, the hex conversion utility bridges these holes by using the address records in the output file to skip ahead to the start of the next section. In other words, there may be discontinuities in the output file addresses. Some EPROM programmers do not support address discontinuities.

In image mode, there are no discontinuities. Each output file contains a continuous stream of data that corresponds exactly to an address range in target memory. Any holes before, between, or after sections are filled with a fill value that you supply.

An output file converted by using image mode still has address records, because many of the hexadecimal formats require an address on each line. However, in image mode, these addresses are always contiguous.

Defining the Ranges of Target Memory

NOTE: If you use image mode, you must also use a ROMS directive. In image mode, each output file corresponds directly to a range of target memory. You must define the ranges. If you do not supply the ranges of target memory, the utility tries to build a memory image of the entire target processor address space. This is 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.

13.9.2 Specifying a Fill Value

The --fill option specifies a value for filling the holes between sections. The fill value must be specified as an integer constant following the --fill option. The width of the constant is assumed to be that of a word on the target processor. For example, specifying --fill=0xFF0. The constant value is not sign extended.

The hex conversion utility uses a default fill value of 0 if you do not specify a value with the fill option. *The --fill option is valid only when you use --image*; otherwise, it is ignored.

13.9.3 Steps to Follow in Using Image Mode

Step 1: Define the ranges of target memory with a ROMS directive. See Section 13.4.

Step 2: Invoke the hex conversion utility with the --image option. You can optionally use the --zero option to reset the address origin to 0 for each output file. If you do not specify a fill value with the ROMS directive and you want a value other than the default of 0, use the --fill option.

13.10 Building a Table for an On-Chip Boot Loader

Some DSP devices, such as the C55x, 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.

13.10.1 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 builds a complete image of the table according to the required format, 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 serially boot a 16-bit TMS320C55x from a single 8-bit EPROM by using the --serial8 --memwidth16 options to configure the width of the boot table. The hex conversion utility automatically adjusts the table's format and length. See the boot loader examples in the various application reports with titles such as *Using the* device name *Bootloader* for an illustration of a boot table.

13.10.2 The Boot Table Format

The boot table format is simple. Typically, there is a header record containing 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 record follows the last block. The application reports for specific C55x devices describe the exact table format.

13.10.3 How to Build the Boot Table

Table 13-2 summarizes the hex conversion utility options available for the boot loader.

Table 13-2. Boot-Loader Options

Option	Description
--boot	Convert all sections into bootable form (use instead of a SECTIONS directive).
--bootorg=*addr*	Specify the source address of the boot-loader table.
--bootpage=*page value*	Specify the page containing the boot-loader routine. The *page* argument tells the hex utility where to place the boot-loader routine.
--delay=*value*	Insert a delay of the given *value* into the configuration section of the boot table.
--entry_point=*value*	Specify the entry point at which to begin execution after boot loading. The *value* can be an address or a global symbol.
--parallel16	Specify a 16-bit parallel interface boot table (−memwidth 16 and −romwidth 16)
--parallel32	Specify a 32-bit parallel interface boot table (−memwidth 16 and −romwidth 32)

Table 13-2. Boot-Loader Options (continued)

Option	Description
--reg_config *addr*,*data*,*data*,...	Insert initialization command into the configuration section of the boot table. Initialize locations starting at location *addr* with the given *data* values. No spaces are permitted in the addr/data argument list.
--serial8	Specify an 8-bit serial interface boot table (−memwidth 8 and− romwidth 8)
--serial16	Specify a 16-bit serial interface boot table (−memwidth 16 and −romwidth 16)
−v*device*	Specify the silicon boot table version

13.10.3.1 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 object file. Uninitialized sections are not converted by the hex conversion utility (see Section 13.5). You must link into your application a boot loader routine that will read the boot table and perform the copy operations. It should be linked to its eventual run-time address.

When you select a section for placement in a boot-loader table, the hex conversion utility places the section's *load address* in the destination address field for the block in the boot table. The section content is then treated as raw data for that block. *The hex conversion utility does not use the section run address.* When linking, you need not worry about the ROM address or the construction of the boot table; the hex conversion utility handles this.

Step 2: **Identify the bootable sections.** You can use the --boot option to tell the hex conversion utility to configure all sections for boot loading. Or, you can use a SECTIONS directive to select specific sections to be configured (see Section 13.5). 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 C55x and booting from memory location 8000h, specify −bootorg=0x8000. The address field for the boot table in the hex conversion utility output file will then start at 0x8000.

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 entry point, parallel interface, or serial interface options as needed. Use the -v option to indicate which boot table format is to be used. The default is -v5510:2, which specifies revision 2 of the format defined for the 5510 device. This format is used on most of the C55x devices. See the application report for the specific bootloader you are using.

If the target device boot loader supports register initialization Use the --reg_config and --delay options to specify any desired initializations. The initializations occur in the order the options are given. Delays are inserted in the sequence according to where the --delay options appear in the list of options.

Step 5: **Describe your system memory configuration.** See Section 13.3 and Section 13.4 for details.

13.10.3.2 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.

13.10.3.3 Setting the Entry Point for the Boot Table

After the boot routine finishes copying data, it branches to the entry point indicated by the object file. By using the --entry_point 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 0x0123 after loading, specify --entry_point=0x0123 on the command line or in a command file. You can determine the --entry_point address by looking at the map file that the linker generates.

Valid Entry Points

NOTE: The value can be a constant, or it can be a symbol that is externally defined (for example, with a .global) in the assembly source.

13.10.4 Booting From a Device Peripheral

You can choose to boot from a serial or parallel port by using the --parallel16, --parallel32, --serial8, or --serial16 option. Your selection of an option depends on the target device and the channel you want to use. For example, to boot a C55x from its 16-bit McBSP port, specify −serial16 on the command line or in a command file. To boot a C55x from one of its EMIF ports, specify --parallel16 or --parallel32.

You need to address certain 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 in Section 13.10.3.1, 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.

13.10.5 Booting From Memory

The C55x can also boot from a boot table in memory. 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.

Example 13-3 allows you to boot the .text section of abc.out from a byte-wide EPROM at location 0x8000.

Example 13-3. Sample Command File for Booting From a C55x EPROM

```
abc.out              /* input file                       */
-outfile=abc.i       /* output file                      */
-intel               /* Intel format                     */
--memwidth=8         /* 8-bit memory                     */
--romwidth=8         /* outfile is bytes, not words      */
--bootorg=0x8000     /* external memory boot             */

SECTIONS { .text: BOOT }
```

13.11 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 sectionand/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.

13.11.1 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.

Non-boot loader mode. The address field of the hex conversion utility output file is controlled by the following mechanisms listed from low to high priority:

1. **The linker command file.** By default, the address field of the hex conversion utility output file is the load address (as given in the linker command file) and the hex conversion utility parameter values. The relationship is summarized as follows, if paddr is not specified:.

   out_file_addr ≡ load_addr × (data_width ÷ mem_width) (1)

out_file_addr	is the address of the output file.
load_addr	is the linker-assigned load address.
data_width	is specified as 8 bits for the TMS320C55x devices. See Section 13.3.2.
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 Section 13.3.3.

   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:

   out_file_addr ≡ paddr_val + (load_addr - sect_beg_load_addr) ×(data_width ÷ mem_width) (2)

out_file_addr	is the address of the output file.
paddr_va	is the value supplied with the paddr parameter inside the SECTIONS directive.
load_addr	is the linker-assigned load address.
sec_beg_load_addr	is the section load address assigned by the linker.
data_width	is specified as 8 bits for the TMS320C55x devices. See Section 13.3.2.
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 Section 13.3.3.

   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 sections that are in the boot table into consecutive memory locations. Each section becomes a boot table block whose destination address is equal to the linker-assigned section load address.

In a boot table, the address field of the hex conversion utility output file is not related to the section load addresses assigned by the linker. The address fields of the boot table are simply offsets to the beginning of the table. The section load addresses assigned by the linker will be encoded into the boot table along with the size of the section and the data contained within the section. These addresses will be used to store the data into memory during the boot load process.

The beginning of the boot table defaults to the linked load address of the first bootable section in the 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.

13.11.2 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.

13.11.3 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. Example 13-4 shows a command file that can be used to avoid the hole at the beginning of .sec1.

  If your file contains multiple sections and one section uses a paddr parameter, then all sections must use the paddr parameter.

Example 13-4. Hex Command File for Avoiding a Hole at the Beginning of a Section

```
--intel
a.out
--map=a.map

ROMS
{
   ROM : org = 0x0100, length = 0x200, romwidth = 8,
         memwidth = 8
}

SECTIONS
{
   sec1: paddr = 0x100
}
```

- **Use the --bootorg option or use the ROMS origin parameter (for boot loading only)**. As described in Section 13.11.1, 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.

13.12 Control Hex Conversion Utility Diagnostics

The hex conversion utility uses certain C/C++ compiler options to control hex-converter-generated diagnostics.

--diag_error=*id*	Categorizes the diagnostic identified by *id* as an error. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_error=*id* to recategorize the diagnostic as an error. You can only alter the severity of discretionary diagnostics.
--diag_remark=*id*	Categorizes the diagnostic identified by *id* as a remark. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_remark=*id* to recategorize the diagnostic as a remark. You can only alter the severity of discretionary diagnostics.
--diag_suppress=*id*	Suppresses the diagnostic identified by *id*. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_suppress=*id* to suppress the diagnostic. You can only suppress discretionary diagnostics.
--diag_warning=*id*	Categorizes the diagnostic identified by *id* as a warning. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_warning=*id* to recategorize the diagnostic as a warning. You can only alter the severity of discretionary diagnostics.
--display_error_number	Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See the *TMS320C55x Optimizing C/C++ Compiler User's Guide* for more information on understanding diagnostic messages.
--issue_remarks	Issues remarks (nonserious warnings), which are suppressed by default.
--no_warnings	Suppresses warning diagnostics (errors are still issued).
--set_error_limit=*count*	Sets the error limit to *count*, which can be any decimal value. The linker abandons linking after this number of errors. (The default is 100.)
--verbose_diagnostics	Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line

13.13 Description of the Object Formats

The hex conversion utility has options that identify each format. Table 13-3 specifies the format options. They are described in the following sections.

- You need to use only one of these options on the command line. If you use more than one option, the last one you list overrides the others.
- The default format is Tektronix (--tektronix option).

Table 13-3. Options for Specifying Hex Conversion Formats

Option	Alias	Format	Address Bits	Default Width
--ascii	-a	ASCII-Hex	16	8
--intel	-i	Intel	32	8
--motorola=1	-m1	Motorola-S1	16	8
--motorola=2	-m2	Motorola-S2	24	8
--motorola=3	-m3	Motorola-S3	32	8
--ti-tagged	-t	TI-Tagged	16	16
--ti_txt		TI_TXT	8	8
--tektronix	-x	Tektronix	32	8

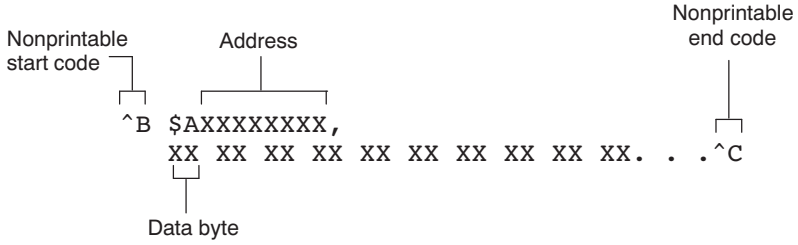
Address bits determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

The **default width** determines the default output width of the format. You can change the default width by using the --romwidth option or by using the romwidth parameter in the ROMS directive. You cannot change the default width of the TI-Tagged format, which supports a 16-bit width only.

13.13.1 ASCII-Hex Object Format (--ascii Option)

The ASCII-Hex object format supports 16-bit addresses. The format consists of a byte stream with bytes separated by spaces. Figure 13-7 illustrates the ASCII-Hex format.

Figure 13-7. ASCII-Hex Object Format



The file begins with an ASCII STX character (ctrl-B, 02h) and ends with an ASCII ETX character (ctrl-C, 03h). Address records are indicated with $AXXXXXXX, in which XXXXXXXX is a 8-digit (16-bit) hexadecimal address. The address records are present only in the following situations:

- When discontinuities occur
- When the byte stream does not begin at address 0

You can avoid all discontinuities and any address records by using the --image and --zero options. This creates output that is simply a list of byte values.

13.13.2 Intel MCS-86 Object Format (--intel 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

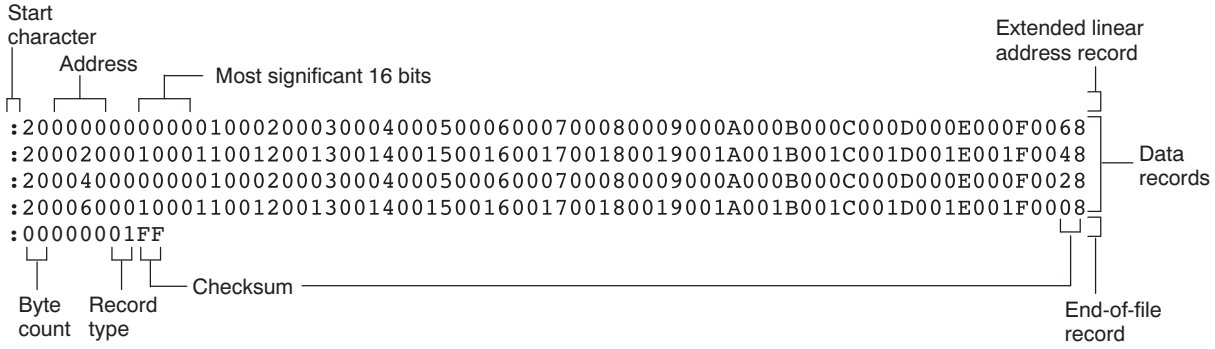
Record type00, the data record, begins with a colon (:) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended linear address) record to create a full 32-bit address. The checksum is the 2s complement (in binary form) of the preceding bytes in the record, including byte count, address, and data bytes.

Record type 01, the end-of-file record, also begins with a colon (:), followed by the byte count, the address, the record type (01), and the checksum.

Record type 04, the extended linear address record, specifies the upper 16 address bits. It begins with a colon (:), followed by the byte count, a dummy address of 0h, the record type (04), the most significant 16 bits of the address, and the checksum. The subsequent address fields in the data records contain the least significant bytes of the address.

Figure 13-8 illustrates the Intel hexadecimal object format.

Figure 13-8. Intel Hexadecimal Object Format



13.13.3 Motorola Exorciser Object Format (--motorola Option)

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 consists of five fields: record type, byte count, address, data, and checksum. The three record types are:

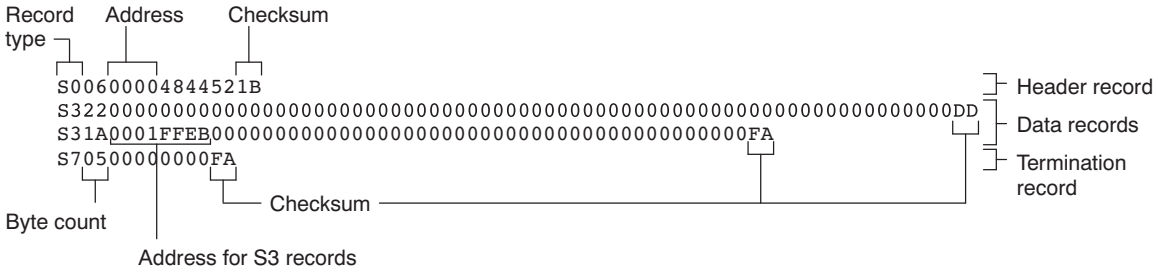
Record Type	Description
S0	Header record
S1	Code/data record for 16-bit addresses (S1 format)
S2	Code/data record for 24-bit addresses (S2 format)
S3	Code/data record for 32-bit addresses (S3 format)
S7	Termination record for 32-bit addresses (S3 format)
S8	Termination record for 24-bit addresses (S2 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 13-9 illustrates the Motorola-S object format.

Figure 13-9. Motorola-S Format



13.13.4 Extended Tektronix Object Format (--tektronix Option)

The Tektronix object format supports 32-bit addresses and has two types of records:

Data records	contains the header field, the load address, and the object code.
Termination records	signifies the end of a module.

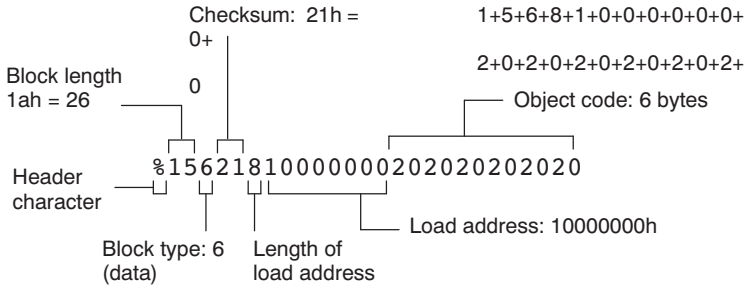
The header field in the data record contains the following information:

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 13-10 illustrates the Tektronix object format.

Figure 13-10. Extended Tektronix Object Format



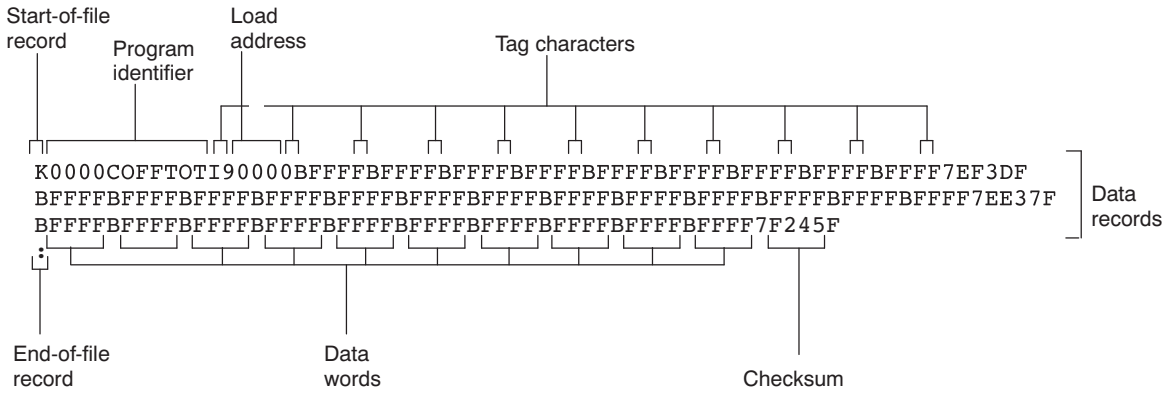
13.13.5 Texas Instruments SDSMAC (TI-Tagged) Object Format (--ti_tagged Option)

The Texas Instruments SDSMAC (TI-Tagged) object format supports 16-bit addresses, including start-of-file record, data records, and end-of-file record. Each data records consists of a series of small fields and is signified by a tag character:

Tag Character	Description
K	Followed by the program identifier
7	Followed by a checksum
8	Followed by a dummy checksum (ignored)
9	Followed by a 16-bit load address
B	Followed by a data word (four characters)
F	Identifies the end of a data record
*	Followed by a data byte (two characters)

Figure 13-11 illustrates the tag characters and fields in TI-Tagged object format.

Figure 13-11. 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 but not required for any data byte. The checksum field, preceded by the tag character 7, is the 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 (:).

13.13.6 TI-TXT Hex Format (--ti_txt Option)

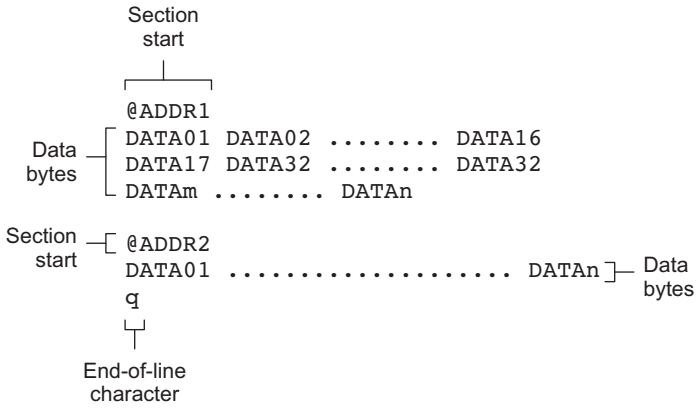
The TI-TXT hex format supports 16-bit hexadecimal data. It consists of section start addresses, data byte, and an end-of-file character. These restrictions apply:

- The number of sections is unlimited.
- Each hexadecimal start address must be even.
- Each line must have 16 data bytes, except the last line of a section.
- Data bytes are separated by a single space.
- The end-of-file termination tag q is mandatory.

The data record contains the following information:

Item	Description
@ADDR	Hexadecimal start address of a section
DATAn	Hexadecimal data byte
q	End-of-file termination character

Figure 13-12. TI-TXT Object Format



Example 13-5. TI-TXT Object Format

```
@F000
31 40 00 03 B2 40 80 5A 20 01 D2 D3 22 00 D2 E3
21 00 3F 40 E8 FD 1F 83 FE 23 F9 3F
@FFFE
00 F0
Q
```

13.14 Hex Conversion Utility Error Messages

section mapped to reserved memory

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 *TMS320C55x DSP CPU Reference 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 13.3 and Section 13.11.

unconfigured memory error

Description This error could have one of two causes:

- The object 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.

Sharing C/C++ Header Files With Assembly Source

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code.

Topic **Page**

14.1 Overview of the .cdecls Directive

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

See the .cdecls directive description for details on the syntax of the .cdecls assembler directive.

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

For example, the following code causes the warning to be issued:

```
.cdecls C,NOLIST
%{
    #define ASMTEST 1
%}

.cdecls C,NOLIST
%{
    #ifndef ASMTEST
        #warn "ASMTEST not defined!"  /* will be issued */
    #endif
%}
```

Therefore, a typical use of the .cdecls block is expected to be a single usage near the beginning of the assembly source file, in which all necessary C/C++ header files are included.

Use the compiler --include_path=*path* options to specify additional include file paths needed for the header files used in assembly, as you would when compiling C files.

Any C/C++ errors or warnings generated by the code of the .cdecls are emitted as they normally would for the C/C++ source code. C/C++ errors cause the directive to fail, and any resulting converted assembly is not included.

C/C++ constructs that cannot be converted, such as function-like macros or variable definitions, cause a comment to be output to the converted assembly file. For example:

```
; ASM HEADER WARNING - variable definition 'ABCD' ignored
```

The prefix ASM HEADER WARNING appears at the beginning of each message. To see the warnings, either the WARN parameter needs to be specified so the messages are displayed on STDERR, or else the LIST parameter needs to be specified so the warnings appear in the listing file, if any.

Finally, note that the converted assembly code does not appear in the same order as the original C/C++ source code and C/C++ constructs may be simplified to a normalized form during the conversion process, but this should not affect their final usage.

14.2 Notes on C/C++ Conversions

The following sections describe C and C++ conversion elements that you need to be aware of when sharing header files with assembly source.

14.2.1 Comments

Comments are consumed entirely at the C level, and do not appear in the resulting converted assembly file.

14.2.2 Conditional Compilation (#if/#else/#ifdef/etc.)

Conditional compilation is handled entirely at the C level during the conversion step. Define any necessary macros either on the command line (using the compiler --define=*name*=*value* option) or within a .cdecls block using #define. The #if, #ifdef, etc. C/C++ directives are **not** converted to assembly .if, .else, .elseif, and .endif directives.

14.2.3 Pragmas

Pragmas found in the C/C++ source code cause a warning to be generated as they are not converted. They have no other effect on the resulting assembly file. See the .cdecls topic for the WARN and NOWARN parameter discussion for where these warnings are created.

14.2.4 The #error and #warning Directives

These preprocessor directives are handled completely by the compiler during the parsing step of conversion. If one of these directives is encountered, the appropriate error or warning message is emitted. These directives are not converted to .emsg or .wmsg in the assembly output.

14.2.5 Predefined symbol _ _ASM_HEADER_ _

The C/C++ macro _ _ASM_HEADER_ _ is defined in the compiler while processing code within .cdecls. This allows you to make changes in your code, such as not compiling definitions, during the .cdecls processing.

Be Careful With the _ _ASM_HEADER_ _ Macro

NOTE: You must be very careful not to use this macro to introduce any changes in the code that could result in inconsistencies between the code processed while compiling the C/C++ source and while converting to assembly.

14.2.6 Usage Within C/C++ asm( ) Statements

The .cdecls directive is not allowed within C/C++ asm( ) statements and will cause an error to be generated.

14.2.7 The #include Directive

The C/C++ #include preprocessor directive is handled transparently by the compiler during the conversion step. Such #includes can be nested as deeply as desired as in C/C++ source. The assembly directives .include and .copy are not used or needed within a .cdecls. Use the command line --include_path option to specify additional paths to be searched for included files, as you would for C compilation.

14.2.8 Conversion of #define Macros

Only object-like macros are converted to assembly. Function-like macros have no assembly representation and so cannot be converted. Pre-defined and built-in C/C++ macros are not converted to assembly (i.e., __FILE__, __TIME__, __TI_COMPILER_VERSION__, etc.). For example, this code is converted to assembly because it is an object-like macro:

```
#define NAME Charley
```

This code is not converted to assembly because it is a function-like macro:

```
#define MAX(x,y) (x>y ? x : y)
```

Some macros, while they are converted, have no functional use in the containing assembly file. For example, the following results in the assembly substitution symbol FOREVER being set to the value while(1), although this has no useful use in assembly because while(1) is not legal assembly code.

```
#define FOREVER while(1)
```

Macro values are **not** interpreted as they are converted. For example, the following results in the assembler substitution symbol OFFSET being set to the literal string value 5+12 and **not** the value 17. This happens because the semantics of the C/C++ language require that macros are evaluated in context and not when they are parsed.

```
#define OFFSET 5+12
```

Because macros in C/C++ are evaluated in their usage context, C/C++ printf escape sequences such as \n are not converted to a single character in the converted assembly macro. See Section 14.2.11 for suggestions on how to use C/C++ macro strings.

Macros are converted using the new .define directive (see Section 14.4.2), which functions similarly to the .asg assembler directive. The exception is that .define disallows redefinitions of register symbols and mnemonics to prevent the conversion from corrupting the basic assembly environment. To remove a macro from the assembly scope, .undef can be used following the .cdecls that defines it (see Section 14.4.3).

The macro functionality of # (stringize operator) is only useful within functional macros. Since functional macros are not supported by this process, # is not supported either. The concatenation operator ## is only useful in a functional context, but can be used degenerately to concatenate two strings and so it is supported in that context.

14.2.9 The #undef Directive

Symbols undefined using the #undef directive before the end of the .cdecls are not converted to assembly.

14.2.10 Enumerations

Enumeration members are converted to .enum elements in assembly. For example:

```
enum state { ACTIVE=0x10, SLEEPING=0x01, INTERRUPT=0x100, POWEROFF, LAST};
```

is converted to the following assembly code:

```
state      .enum
ACTIVE     .emember 16
SLEEPING   .emember 1
NTERRUPT   .emember 256
POWEROFF   .emember 257
LAST       .emember 258
           .endenum
```

The members are used via the pseudo-scoping created by the .enum directive.

The usage is similar to that for accessing structure members, enum_name.member.

This pseudo-scoping is used to prevent enumeration member names from corrupting other symbols within the assembly environment.

14.2.11 C Strings

Because C string escapes such as \n and \t are not converted to hex characters 0x0A and 0x09 until their use in a string constant in a C/C++ program, C macros whose values are strings cannot be represented as expected in assembly substitution symbols. For example:

```
#define MSG "\tHI\n"
```

becomes, in assembly:

```
.define """\tHI\n""",MSG ; 6 quoted characters! not 5!
```

When used in a C string context, you expect this statement to be converted to 5 characters (tab, H, I, newline, NULL), but the .string assembler directive does not know how to perform the C escape conversions.

You can use the .cstring directive to cause the escape sequences and NULL termination to be properly handled as they would in C/C++. Using the above symbol MSG with a .cstring directive results in 5 characters of memory being allocated, the same characters as would result if used in a C/C++ strong context. (See Section 14.4.7 for the .cstring directive syntax.)

14.2.12 C/C++ Built-In Functions

The C/C++ built-in functions, such as sizeof( ), are not translated to their assembly counterparts, if any, if they are used in macros. Also, their C expression values are not inserted into the resulting assembly macro because macros are evaluated in context and there is no active context when converting the macros to assembly.

Suitable functions such as $sizeof( ) are available in assembly expressions. However, as the basic types such as int/char/float have no type representation in assembly, there is no way to ask for $sizeof(int), for example, in assembly.

14.2.13 Structures and Unions

C/C++ structures and unions are converted to assembly .struct and .union elements. Padding and ending alignments are added as necessary to make the resulting assembly structure have the same size and member offsets as the C/C++ source. The primary purpose is to allow access to members of C/C++ structures, as well as to facilitate debugging of the assembly code. For nested structures, the assembly .tag feature is used to refer to other structures/unions.

The alignment is also passed from the C/C++ source so that the assembly symbol is marked with the same alignment as the C/C++ symbol. (See Section 14.2.3 for information about pragmas, which may attempt to modify structures.) Because the alignment of structures is stored in the assembly symbol, built-in assembly functions like $sizeof( ) and $alignof( ) can be used on the resulting structure name symbol.

When using unnamed structures (or unions) in typedefs, such as:

```
typedef struct { int a_member; } mystrname;
```

This is really a shorthand way of writing:

```
struct temporary_name { int a_member; };
typedef temporary_name mystrname;
```

The conversion processes the above statements in the same manner: generating a temporary name for the structure and then using .define to output a typedef from the temporary name to the user name. You should use your *mystrname* in assembly the same as you would in C/C++, but do not be confused by the assembly structure definition in the list, which contains the temporary name. You can avoid the temporary name by specifying a name for the structure, as in:

```
typedef struct a_st_name { ... } mystrname;
```

If a shorthand method is used in C to declare a variable with a particular structure, for example:

```
extern struct a_name { int a_member; } a_variable;
```

Then after the structure is converted to assembly, a .tag directive is generated to declare the structure of the external variable, such as:

```
_a_variable .tag a_st_name
```

This allows you to refer to _a_variable.a_member in your assembly code.

14.2.14 Function/Variable Prototypes

Non-static function and variable prototypes (not definitions) will result in a .global directive being generated for each symbol found.

See Section 14.3.1 for C++ name mangling issues.

Function and variable definitions will result in a warning message being generated (see the WARN/NOWARN parameter discussion for where these warnings are created) for each, and they will not be represented in the converted assembly.

The assembly symbol representing the variable declarations will not contain type information about those symbols. Only a .global will be issued for them. Therefore, it is your responsibility to ensure the symbol is used appropriately.

See Section 14.2.13 for information on variables names which are of a structure/union type.

14.2.15 C Constant Suffixes

The C constant suffixes u, l, and f are passed to the assembly unchanged. The assembler will ignore these suffixes if used in assembly expressions.

14.2.16 Basic C/C++ Types

Only complex types (structures and unions) in the C/C++ source code are converted to assembly. Basic types such as int, char, or float are not converted or represented in assembly beyond any existing .int, .char, .float, etc. directives that previously existed in assembly.

Typedefs of basic types are therefore also not represented in the converted assembly.

14.3 Notes on C++ Specific Conversions

The following sections describe C++ specific conversion elements that you need to be aware of when sharing header files with assembly source.

14.3.1 Name Mangling

Symbol names may be mangled in C++ source files. When mangling occurs, the converted assembly will use the mangled names to avoid symbol name clashes. You can use the demangler (dem55) to demangle names and identify the correct symbols to use in assembly.

To defeat name mangling in C++ for symbols where polymorphism (calling a function of the same name with different kinds of arguments) is not required, use the following syntax:

```
extern "C" void somefunc(int arg);
```

The above format is the short method for declaring a single function. To use this method for multiple functions, you can also use the following syntax:

```
extern "C"
  {
      void somefunc(int arg);
      int  anotherfunc(int arg);
      ...
   }
```

14.3.2 Derived Classes

Derived classes are only partially supported when converting to assembly because of issues related to C++ scoping which does not exist in assembly. The greatest difference is that base class members do not automatically become full (top-level) members of the derived class. For example:

```
-------------------------------------------------------
        class base
        {
           public:
                int b1;
        };

        class derived : public base
        {
            public:
               int d1;
        }
```

In C++ code, the class derived would contain both integers b1 and d1. In the converted assembly structure "derived", the members of the base class must be accessed using the name of the base class, such as derived.__b_base.b1 rather than the expected derived.b1.

A non-virtual, non-empty base class will have __b_ prepended to its name within the derived class to signify it is a base class name. That is why the example above is derived.__b_base.b1 and not simply derived.base.b1.

14.3.3 Templates

No support exists for templates.

14.3.4 Virtual Functions

No support exists for virtual functions, as they have no assembly representation.

14.4 Special Assembler Support

14.4.1 Enumerations (.enum/.emember/.endenum)

New directives have been created to support a pseudo-scoping for enumerations.

The format of these new directives is:

ENUM_NAME	**.enum**
MEMBER1	**.emember** [*value*]
MEMBER2	**.emember** [*value*]
...	
	.endenum

The **.enum** directive begins the enumeration definition and **.endenum** terminates it.

The enumeration name (*ENUM_NAME*) cannot be used to allocate space; its size is reported as zero.

The format to use the value of a member is *ENUM_NAME.MEMBER*, similar to a structure member usage.

The **.emember** directive optionally accepts the value to set the member to, just as in C/C++. If not specified, the member takes a value one more than the previous member. As in C/C++, member names cannot be duplicated, although values can be. Unless specified with .emember, the first enumeration member will be given the value 0 (zero), as in C/C++.

The .endenum directive cannot be used with a label, as structure .endstruct directives can, because the .endenum directive has no value like the .endstruct does (containing the size of the structure).

Conditional compilation directives (.if/.else/.elseif/.endif) are the only other non-enumeration code allowed within the .enum/.endenum sequence.

14.4.2 The .define Directive

The new .define directive functions in the same manner as the .asg directive, except that .define disallows creation of a substitution symbol that has the same name as a register symbol or mnemonic. It does not create a new symbol name space in the assembler, rather it uses the existing substitution symbol name space. The syntax for the directive is:

.define *substitution string* **,** *substitution symbol name*

The .define directive is used to prevent corruption of the assembly environment when converting C/C++ headers.

14.4.3 The .undefine/.unasg Directives

The .undef directive is used to remove the definition of a substitution symbol created using .define or .asg. This directive will remove the named symbol from the substitution symbol table from the point of the .undef to the end of the assembly file. The syntax for these directives is:

.undefine *substitution symbol name*

.unasg *substitution symbol name*

This can be used to remove from the assembly environment any C/C++ macros that may cause a problem.

Also see Section 14.4.2, which covers the .define directive.

14.4.4 The $defined( ) Built-In Function

The $defined directive returns true/1 or false/0 depending on whether the name exists in the current substitution symbol table or the standard symbol table. In essence $defined returns TRUE if the assembler has any user symbol in scope by that name. This differs from $isdefed in that $isdefed only tests for NON-substitution symbols. The syntax is:

$defined(*substitution symbol name* **)**

A statement such as ".if $defined(macroname)" is then similar to the C code "#ifdef macroname".

See Section 14.4.2 and Section 14.4.3 for the use of .define and .undef in assembly.

14.4.5 The $sizeof Built-In Function

The new assembly built-in function $sizeof( ) can be used to query the size of a structure in assembly. It is an alias for the already existing $structsz( ). The syntax is:

$sizeof(*structure name* **)**

The $sizeof function can then be used similarly to the C built-in function sizeof( ).

The assembler's $sizeof( ) built-in function cannot be used to ask for the size of basic C/C++ types, such as $sizeof(int), because those basic type names are not represented in assembly. Only complex types are converted from C/C++ to assembly.

Also see Section 14.2.12, which notes that this conversion does not happen automatically if the C/C++ sizeof( ) built-in function is used within a macro.

14.4.6 Structure/Union Alignment and $alignof( )

The assembly .struct and .union directives now take an optional second argument which can be used to specify a minimum alignment to be applied to the symbol name. This is used by the conversion process to pass the specific alignment from C/C++ to assembly.

The assembly built-in function $alignof( ) can be used to report the alignment of these structures. This can be used even on assembly structures, and the function will return the minimum alignment calculated by the assembler.

14.4.7 The .cstring Directive

You can use the new .cstring directive to cause the escape sequences and NULL termination to be properly handled as they would in C/C++.

```
.cstring "String with C escapes.\nWill be NULL terminated.\012"
```

See Section 14.2.11 for more information on the new .cstring directive.

Symbolic Debugging Directives

The assembler supports several directives that the TMS320C55x C/C++ compiler uses for symbolic debugging. These directives differ for the two debugging formats, DWARF and COFF.

These directives are not meant for use by assembly-language programmers. They require arguments that can be difficult to calculate manually, and their usage must conform to a predetermined agreement between the compiler, the assembler, and the debugger. This appendix documents these directives for informational purposes only.

Topic **Page**

A.1 DWARF Debugging Format

A subset of the DWARF symbolic debugging directives are always listed in the assembly language file that the compiler creates for program analysis purposes. To list the complete set used for full symbolic debug, invoke the compiler with the --symdebug:dwarf option, as shown below:

```
cl55 --symdebug:dwarf --keep_asm input_file
```

The --keep_asm option instructs the compiler to retain the generated assembly file.

To disable the generation of all symbolic debug directives, invoke the compiler with the -symdebug:none option:

```
cl55 --symdebug:none --keep_asm input_file
```

The DWARF debugging format consists of the following directives:

- The **.dwtag** and **.dwendtag** directives define a Debug Information Entry (DIE) in the .debug_info section.
- The **.dwattr** directive adds an attribute to an existing DIE.
- The **.dwpsn** directive identifies the source position of a C/C++ statement.
- The **.dwcie** and **.dwendentry** directives define a Common Information Entry (CIE) in the .debug_frame section.
- The **.dwfde** and **.dwendentry** directives define a Frame Description Entry (FDE) in the .debug_frame section.
- The **.dwcfi** directive defines a call frame instruction for a CIE or FDE.

A.2 COFF Debugging Format

COFF symbolic debug is now obsolete. These directives are supported for backwards-compatibility only. The decision to switch to DWARF as the symbolic debug format was made to overcome many limitations of COFF symbolic debug, including the absence of C++ support.

The COFF debugging format consists of the following directives:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the variable or function.
- The **.stag**, **.etag**, and **.utag** directives define structures, enumerations, and unions, respectively. The **.member** directive specifies a member of a structure, enumeration, or union. The **.eos** directive ends a structure, enumeration, or union definition.
- The **.func** and **.endfunc** directives specify the beginning and ending lines of a C/C++ function.
- The **.block** and **.endblock** directives specify the bounds of C/C++ blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source filename.
- The **.line** directive identifies the line number of a C/C++ source statement.

A.3 Debug Directive Syntax

Table A-1 is an alphabetical listing of the symbolic debugging directives. For information on the C/C++ compiler, refer to the *TMS320C55x Optimizing C/C++ Compiler User's Guide.*

Table A-1. Symbolic Debugging Directives

Label	Directive	Arguments
	.block	[*beginning line number*]
	.dwattr	*DIE label* **,** *DIE attribute name* **(** *DIE attribute value* **)[,** *DIE attribute name* **(** *attribute value* **)** [**,** ...]
	.dwcfi	*call frame instruction opcode*[**,** *operand*[**,** *operand*]]
CIE label	**.dwcie**	*version* **,** *return address register*
	.dwendentry	
	.dwendtag	
	.dwfde	*CIE label*
	.dwpsn	" *filename* "**,** *line number* **,** *column number*
DIE label	.**dwtag**	*DIE tag name* **,** *DIE attribute name* **(** *DIE attribute value* **)[,** *DIE attribute name* **(** *attribute value* **)** [**,** ...]
	.endblock	[*ending line number*]
	.endfunc	[*ending line number*[**,** *register mask*[**,** *frame size*]]]
	.eos	
	.**etag**	*name*[**,** *size*]
	.file	" *filename* "
	.func	[*beginning line number*]
	.**line**	*line number*[**,** *address*]
	.member	*name* **,** *value*[**,** *type* **,** *storage class* **,** *size* **,** *tag* **,** *dims*]
	.stag	*name*[**,** *size*]
	.sym	*name* **,** *value*[**,** *type* **,** *storage class* **,** *size* **,** *tag* **,** *dims*]
	.utag	*name*[**,** *size*]

XML Link Information File Description

The TMS320C55x linker supports the generation of an XML link information file via the --xml_link_info *file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker-generated map file.

As the linker evolves, the XML link information file may be extended to include additional information that could be useful for static analysis of linker results.

This appendix enumerates all of the elements that are generated by the linker into the XML link information file.

Topic **Page**

B.1 XML Information File Element Types

These element types will be generated by the linker:

- **Container elements** represent an object that contains other elements that describe the object. Container elements have an id attribute that makes them accessible from other elements.
- **String elements** contain a string representation of their value.
- **Constant elements** contain a 32-bit unsigned long representation of their value (with a 0x prefix).
- **Reference elements** are empty elements that contain an idref attribute that specifies a link to another container element.

In Section B.2, the element type is specified for each element in parentheses following the element description. For instance, the <link_time> element lists the time of the link execution (string).

B.2 Document Elements

The root element, or the document element, is **<link_info>**. All other elements contained in the XML link information file are children of the <link_info> element. The following sections describe the elements that an XML information file can contain.

B.2.1 Header Elements

The first elements in the XML link information file provide general information about the linker and the link session:

- The **<banner>** element lists the name of the executable and the version information (string).
- The **<copyright>** element lists the TI copyright information (string).
- The **<link_time>** is a timestamp representation of the link time (unsigned 32-bit int).
- The **<output_file>** element lists the name of the linked output file generated (string).
- The **<entry_point>** element specifies the program entry point, as determined by the linker (container) with two entries:
  - The **<name>** is the entry point symbol name, if any (string).
  - The **<address>** is the entry point address (constant).

Example B-1. Header Element for the hi.out Output File

```
<banner>TMS320Cxx Linker           Version x.xx (Jan 6 2008)</banner>
<copyright>Copyright (c) 1996-2008 Texas Instruments Incorporated</copyright>
<link_time>0x43dfd8a4</link_time>
<output_file>hi.out</output_file>
<entry_point>
   <name>_c_int00</name>
   <address>0xaf80</address>
</entry_point>
```

B.2.2 Input File List

The next section of the XML link information file is the input file list, which is delimited with a **<input_file_list>** container element. The <input_file_list> can contain any number of <input_file> elements.

Each **<input_file>** instance specifies the input file involved in the link. Each <input_file> has an id attribute that can be referenced by other elements, such as an <object_component>. An <input_file> is a container element enclosing the following elements:

- The **<path>** element names a directory path, if applicable (string).
- The **<kind>** element specifies a file type, either archive or object (string).
- The **<file>** element specifies an archive name or filename (string).
- The **<name>** element specifies an object file name, or archive member name (string).

Example B-2. Input File List for the hi.out Output File

```
<input_file_list>
   <input_file id="fl-1">
      <kind>object</kind>
      <file>hi.obj</file>
      <name>hi.obj</name>
   </input_file>
   <input_file id="fl-2">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>boot.obj</name>
   </input_file>
   <input_file id="fl-3">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>exit.obj</name>
   </input_file>
   <input_file id="fl-4">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>printf.obj</name>
   </input_file>
...
</input_file_list>
```

B.2.3 Object Component List

The next section of the XML link information file contains a specification of all of the object components that are involved in the link. An example of an object component is an input section. In general, an object component is the smallest piece of object that can be manipulated by the linker.

The **<object_component_list>** is a container element enclosing any number of <object_component> elements.

Each **<object_component>** specifies a single object component. Each <object_component> has an id attribute so that it can be referenced directly from other elements, such as a <logical_group>. An <object_component> is a container element enclosing the following elements:

- The **<name>** element names the object component (string).
- The **<load_address>** element specifies the load-time address of the object component (constant).
- The **<run_address>** element specifies the run-time address of the object component (constant).
- The **<size>** element specifies the size of the object component (constant).
- The **<input_file_ref>** element specifies the source file where the object component originated (reference).

Example B-3. Object Component List for the fl-4 Input File

```
<object_component id="oc-20">
   <name>.text</name>
   <load_address>0xac00</load_address>
   <run_address>0xac00</run_address>
   <size>0xc0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
   <name>.data</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
   <name>.bss</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
```

B.2.4 Logical Group List

The <**logical_group_list**> section of the XML link information file is similar to the output section listing in a linker-generated map file. However, the XML link information file contains a specification of GROUP and UNION output sections, which are not represented in a map file. There are three kinds of list items that can occur in a <logical_group_list>:

- The <**logical_group**> is the specification of a section or GROUP that contains a list of object components or logical group members. Each <logical_group> element is given an id so that it may be referenced from other elements. Each <logical_group> is a container element enclosing the following elements:
  - The <**name**> element names the logical group (string).
  - The <**load_address**> element specifies the load-time address of the logical group (constant).
  - The <**run_address**> element specifies the run-time address of the logical group (constant).
  - The <**size**> element specifies the size of the logical group (constant).
  - The <**contents**> element lists elements contained in this logical group (container). These elements refer to each of the member objects contained in this logical group:
    - The <**object_component_ref**> is an object component that is contained in this logical group (reference).
    - The <**logical_group_ref**> is a logical group that is contained in this logical group (reference).
- The <**overlay**> is a special kind of logical group that represents a UNION, or a set of objects that share the same memory space (container). Each <overlay> element is given an id so that it may be referenced from other elements (like from an <allocated_space> element in the placement map). Each <overlay> contains the following elements:
  - The <**name**> element names the overlay (string).
  - The <**run_address**> element specifies the run-time address of overlay (constant).
  - The <**size**> element specifies the size of logical group (constant).
  - The <**contents**> container element lists elements contained in this overlay. These elements refer to each of the member objects contained in this logical group:
    - The <**object_component_ref**> is an object component that is contained in this logical group (reference).
    - The <**logical_group_ref**> is a logical group that is contained in this logical group (reference).
- The <**split_section**> is another special kind of logical group that represents a collection of logical groups that is split among multiple memory areas. Each <split_section> element is given an id so that it may be referenced from other elements. The id consists of the following elements.
  - The <**name**> element names the split section (string).
  - The <**contents**> container element lists elements contained in this split section. The <logical_group_ref> elements refer to each of the member objects contained in this split section, and each element referenced is a logical group that is contained in this split section (reference).

Example B-4. Logical Group List for the fl-4 Input File

```
<logical_group_list>
   ...
      <logical_group id="lg-7">
      <name>.text</name>
      <load_address>0x20</load_address>
      <run_address>0x20</run_address>
      <size>0xb240</size>
      <contents>
         <object_component_ref idref="oc-34"/>
         <object_component_ref idref="oc-108"/>
         <object_component_ref idref="oc-e2"/>
      ...
      </contents>
   </logical_group>
   ...
   <overlay id="lg-b">
      <name>UNION_1</name>
      <run_address>0xb600</run_address>
      <size>0xc0</size>
      <contents>
         <object_component_ref idref="oc-45"/>
         <logical_group_ref idref="lg-8"/>
      </contents>
   </overlay>
    ...
   <split_section id="lg-12">
      <name>.task_scn</name>
      <size>0x120</size>
      <contents>
         <logical_group_ref idref="lg-10"/>
         <logical_group_ref idref="lg-11"/>
      </contents>
   ...
</logical_group_list>
```

B.2.5 Placement Map

The **<placement_map>** element describes the memory placement details of all named memory areas in the application, including unused spaces between logical groups that have been placed in a particular memory area.

The **<memory_area>** is a description of the placement details within a named memory area (container). The description consists of these items:

- The **<name>** names the memory area (string).
- The **<page_id>** gives the id of the memory page in which this memory area is defined (constant).
- The **<origin>** specifies the beginning address of the memory area (constant).
- The **<length>** specifies the length of the memory area (constant).
- The **<used_space>** specifies the amount of allocated space in this area (constant).
- The **<unused_space>** specifies the amount of available space in this area (constant).
- The **<attributes>** lists the RWXI attributes that are associated with this area, if any (string).
- The **<fill_value>** specifies the fill value that is to be placed in unused space, if the fill directive is specified with the memory area (constant).
- The **<usage_details>** lists details of each allocated or available fragment in this memory area. If the fragment is allocated to a logical group, then a <logical_group_ref> element is provided to facilitate access to the details of that logical group. All fragment specifications include <start_address> and <size> elements.
  - The **<allocated_space>** element provides details of an allocated fragment within this memory area (container):
    - The **<start_address>** specifies the address of the fragment (constant).
    - The **<size>** specifies the size of the fragment (constant).
    - The **<logical_group_ref>** provides a reference to the logical group that is allocated to this fragment (reference).
  - The **<available_space** element provides details of an available fragment within this memory area (container):
    - The **<start_address>** specifies the address of the fragment (constant).
    - The **<size>** specifies the size of the fragment (constant).

Example B-5. Placement Map for the fl-4 Input File

```
<placement_map>
   <memory_area>
      <name>PMEM</name>
      <page_id>0x0</page_id>
      <origin>0x20</origin>
      <length>0x100000</length>
      <used_space>0xb240</used_space>
      <unused_space>0xf4dc0</unused_space>
      <attributes>RWXI</attributes>
      <usage_details>
         <allocated_space>
            <start_address>0x20</start_address>
            <size>0xb240</size>
            <logical_group_ref idref="lg-7"/>
         </allocated_space>
         <available_space>
            <start_address>0xb260</start_address>
            <size>0xf4dc0</size>
         </available_space>
      </usage_details>
   </memory_area>
   ...
</placement_map>
```

B.2.6 Far Call Trampoline List

The **<far_call_trampoline_list>** is a list of <far_call_trampoline> elements. The linker supports the generation of far call trampolines to help a call site reach a destination that is out of range. A far call trampoline function is guaranteed to reach the called function (callee) as it may utilize an indirect call to the called function.

The <far_call_trampoline_list> enumerates all of the far call trampolines that are generated by the linker for a particular link. The <far_call_trampoline_list> can contain any number of <far_call_trampoline> elements. Each **<far_call_trampoline>** is a container enclosing the following elements:

- The **<callee_name>** element names the destination function (string).
- The **<callee_address>** is the address of the called function (constant).
- The **<trampoline_object_component_ref>** is a reference to an object component that contains the definition of the trampoline function (reference).
- The **<trampoline_address>** is the address of the trampoline function (constant).
- The **<caller_list>** enumerates all call sites that utilize this trampoline to reach the called function (container).
- The **<trampoline_call_site>** provides the details of a trampoline call site (container) and consists of these items:
  - The **<caller_address>** specifies the call site address (constant).
  - The **<caller_object_component_ref>** is the object component where the call site resides (reference).

Example B-6. Fall Call Trampoline List for the fl-4 Input File

```
<far_call_trampoline_list>
...
   <far_call_trampoline>
      <callee_name>_foo</callee_name>
      <callee_address>0x08000030</callee_address>
      <trampoline_object_component_ref idref="oc-123"/>
      <trampoline_address>0x2020</trampoline_address>
      <caller_list>
         <call_site>
            <caller_address>0x1800</caller_address>
            <caller_object_component_ref idref="oc-23"/>
         </call_site>
         <call_site>
            <caller_address>0x1810</caller_address>
            <caller_object_component_ref idref="oc-23"/>
         </call_site>
      </caller_list>
</far_call_trampoline>
...
</far_call_trampoline_list>
```

B.2.7 Symbol Table

The **<symbol_table>** contains a list of all of the global symbols that are included in the link. The list provides information about a symbol's name and value. In the future, the symbol_table list may provide type information, the object component in which the symbol is defined, storage class, etc.

The **<symbol>** is a container element that specifies the name and value of a symbol with these elements:

- The **<name>** element specifies the symbol name (string).
- The **<value>** element specifies the symbol value (constant).

Example B-7. Symbol Table for the fl-4 Input File

```
<symbol_table>
   <symbol>
      <name>_c_int00</name>
      <value>0xaf80</value>
   </symbol>
   <symbol>
      <name>_main</name>
      <value>0xb1e0</value>
   </symbol>
   <symbol>
      <name>_printf</name>
      <value>0xac00</value>
   </symbol>
   ...
</symbol_table>
```

Glossary

ABI— Application binary interface.

absolute address— An address that is permanently assigned to a TMS320C55x memory location.

absolute lister— A debugging tool that allows you to create assembler listings that contain absolute addresses.

alignment— A process in which the linker places an output section at an address that falls on an *n*-byte boundary, where *n* is a power of 2. You can specify alignment with the SECTIONS linker directive.

allocation— A process in which the linker calculates the final memory addresses of output sections.

ANSI— American National Standards Institute; an organization that establishes standards voluntarily followed by industries.

archive library— A collection of individual files grouped into a single file by the archiver.

archiver— A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.

ASCII— American Standard Code for Information Interchange; 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 definitions. 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.

big endian— An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

binding— A process in which you specify a distinct address for an output section or a symbol.

block— A set of statements that are grouped together within braces and treated as an entity.

.bss section— One of the default object file sections. You use the assembler .bss directive to reserve a specified amount of space in the memory map that you can use later for storing data. The .bss section is uninitialized.

byte— Per ANSI/ISO C, the smallest addressable unit that can hold a character.

C/C++ compiler— A software program that translates C source statements into assembly language source statements.

COFF— Common object file format; a system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

command file— A file that contains options, filenames, directives, or commands for the linker or hex conversion utility.

comment— A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

compiler program— A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

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 type whose value cannot change.

cross-reference lister— A utility that produces an output file that lists the symbols that were defined, what file they were defined in, what reference type they are, what line they were defined on, which lines referenced them, and their assembler and linker final values. The cross-reference lister uses linked object files as input.

cross-reference listing— 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.

.data section— One of the default object file 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).

ELF— Executable and linking format; a system of object files configured according to the System V Application Binary Interface specification.

emulator— A hardware development system that duplicates the TMS320C55x operation.

entry point— A point in target memory where execution starts.

environment variable— A system symbol that you define and assign to a string. Environmental variables are often included in Windows batch files or UNIX shell scripts such as .cshrc or .profile.

epilog— The portion of code in a function that restores the stack and returns.

executable module— A linked object file that can be executed in a target 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 defined or declared in a different program module.

field— For the TMS320C55x, a software-configurable data type whose length can be programmed to be any value in the range of 1-16 bits.

global symbol— A symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.

GROUP— An option of the SECTIONS directive that forces specified output sections to be allocated contiguously (as a group).

hex conversion utility— A utility that converts object files 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 code.

incremental linking— Linking files in several passes. Incremental linking is useful for large applications, because you can partition the application, link the parts separately, and then link all of the parts together.

initialization at load time— An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke it with the --ram_model link option. This method initializes variables at load time instead of run time.

initialized section— A section from an object file that will be linked into an executable module.

input section— A section from an object file that will be linked into an executable module.

ISO— International Organization for Standardization; a worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.

label— A symbol that begins in column 1 of an assembler source statement and corresponds to the address of that statement. A label is the only assembler statement that can begin in column 1.

linker— A software program that combines object files to form an object module that can be allocated into 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 section program counter (SPC).

little endian— An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

loader— A device that places an executable module into system memory.

macro— A user-defined routine that can be used as an instruction.

macro call— The process of invoking a macro.

macro definition— A block of source statements that define the name and the code that make up a macro.

macro expansion— The process of inserting source statements into your code in place of a macro call.

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.

map file— An output file, created by the linker, that shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.

member— The elements or variables of a structure, union, archive, or enumeration.

memory map— A map of target system memory space that is partitioned into functional blocks.

mnemonic— An instruction name that the assembler translates into machine code.

model statement— Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.

named section— An initialized section that is defined with a .sect directive.

object file— An assembled or linked file that contains machine-language object code.

object library— An archive library made up of individual object files.

object module— A linked, executable object file that can be downloaded and executed on a target system.

operand— An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

optimizer— A software tool that improves the execution speed and reduces the size of C programs.

options— Command-line parameters that allow you to request additional or specific functions when you invoke a software tool.

output module— A linked, executable object file that is 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.

partial linking— Linking files in several passes. Incremental linking is useful for large applications because you can partition the application, link the parts separately, and then link all of the parts together.

quiet run— An option that suppresses the normal banner and the progress information.

raw data— Executable code or initialized data in an output section.

relocation— A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

ROM width— The width (in bits) of each output file, or, more specifically, the width of a single data value in the hex conversion utility 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.

run-time-support library— A library file, rts.src, that contains the source for the run time-support functions.

section— A relocatable block of code or data that ultimately will be contiguous with other sections in the memory map.

section program counter (SPC)— An element that keeps track of the current location within a section; each section has its own SPC.

sign extend— A process that fills the unused MSBs of a value with the value's sign bit.

simulator— A software development system that simulates TMS320C55x operation.

source file— A file that contains C/C++ code or assembly language code that is compiled or assembled to form an object file.

static variable— A variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is reentered.

storage class— An entry in the symbol table that indicates how to access a symbol.

string table— A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead they are stored in the string table). The name portion of the symbol's entry points to the location of the string in the string table.

structure— A collection of one or more variables grouped together under a single name.

subsection— A relocatable block of code or data that ultimately will occupy continuous space in the memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.

symbol— A string of alphanumeric characters that represents an address or a value.

symbolic debugging— The ability of a software tool to retain symbolic information that can be used by a debugging tool such as a simulator or an emulator.

tag— An optional *type* name that can be assigned to a structure, union, or enumeration.

target memory— Physical memory in a system into which executable object code is loaded.

.text section— One of the default object file sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.

unconfigured memory— Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section— A object file section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss and .usect directives.

UNION— An option of the SECTIONS directive that causes the linker to allocate the same address to multiple sections.

union— A variable that can hold objects of different types and sizes.

unsigned value— A value that is treated as a nonnegative number, regardless of its actual sign.

variable— A symbol representing a quantity that can assume any of a set of values.

well-defined expression— A term or group of terms that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

word— A 16-bit addressable location in target memory

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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

Products		**Applications**	
Audio	www.ti.com/audio	Communications and Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers and Peripherals	www.ti.com/computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy and Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation and Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
	TI E2E Community Home Page		e2e.ti.com

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