TMS320C28x Assembly Language Tools v19.6.0.STS

User's Guide

Literature Number: SPRU513S
August 2001–Revised June 2019

Contents

List of Figures

List of Tables

Read This First

About This Manual

The *TMS320C28x Assembly Language Tools User's Guide* explains how to use the following Texas Instruments Code Generation object file 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 object file and assembly language tools designed specifically for the TMS320C28x™ 16-bit devices. This book consists of four parts:

- **Introductory information**, consisting of Chapter 1 through Chapter 3, gives you an overview of the object file and assembly language development tools. Chapter 2, in particular, explains object modules and how they can be managed to help your TMS320C28x application load and run. It is highly recommended that developers become familiar with what object modules are and how they are used before using the assembler and linker.
- **Assembler description**, consisting of Chapter 4 through Chapter 6, contains detailed information about using the assembler. Chapter 4 and Chapter 5 explain how to invoke the assembler and discuss source statement format, valid constants and expressions, assembler output, and assembler directives. Chapter 6 focuses on the macro language.
- **Linker and other object file tools description**, consisting of Chapter 7 through Chapter 12, describes in detail each of the tools provided with the assembler to help you create executable object files. Chapter 7 provides details about using the archiver to create object libraries. Chapter 8 explains how to invoke the linker, how the linker operates, and how to use linker directives. Chapter 11 provides a brief overview of some of the object file utilities that can be useful in examining the content of object files as well as removing symbol and debug information to reduce the size of a given object file. Chapter 12 explains how to use the hex conversion utility.
- **Additional Reference material**, consisting of Appendix A through Appendix D, provides supplementary information including symbolic debugging directives used by the TMS320C28x C/C++ compiler. A description of the XML link information file and a glossary are also provided.

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 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:

cl2000 [*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:

cl2000 --run_linker {**--rom_model | --ram_model**} *filenames*
[**--output_file=** *name.out*] **--library=** *libraryname*

- In assembler syntax statements, The leftmost character position, 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*].

.byte $parameter_1$[, ... , $parameter_n$]

- The TMS320C2800 core is referred to as TMS320C28x or C28x.
- Other symbols and abbreviations used throughout this document include the following:

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

See the following resources for further information about the TI Code Generation Tools:

- Texas Instruments Wiki: Compiler topics
- Texas Instruments E2E Community: Compiler forum

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

SPRU514 —TMS320C28x Optimizing C/C++ Compiler User's Guide. Describes the TMS320C28x C/C++ compiler. This C/C++ compiler accepts ANSI standard C/C++ source code and produces assembly language source code for the TMS320C28x devices.

SPRU127 —TMS320C2xx User's Guide. Discusses the hardware aspects of the TMS320C2xx 16-bit fixed-point digital signal processors. It describes the architecture, the instruction set, and the on-chip peripherals.

SPRU430 —TMS320C28x DSP CPU and Instruction Set Reference Guide. Describes the central processing unit (CPU) and the assembly language instructions of the TMS320C28x fixed-point CPU. It also describes emulation features available on these devices.

SPRU566 —TMS320x28xx, 28xxx DSP Peripherals Reference Guide. Describes all the peripherals available for TMS320x28xx and TMS320x28xxx devices.

SPRUEO2 —TMS320C28x Floating Point Unit and Instruction Set Reference Guide. Describes the CPU architecture, pipeline, instruction set, and interrupts of the C28x floating-point DSP.

SPRAC71— TMS320C28x Embedded Application Binary Interface Application Report. Provides a specification for the ELF-based Embedded Application Binary Interface (EABI) for the TMS320C28x family of processors from Texas Instruments. The EABI defines the low-level interface between programs, program components, and the execution environment, including the operating system if one is present.

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.

Trademarks

TMS320C28x is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

Introduction to the Software Development Tools

The TMS320C28x™ 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 TMS320C28x 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

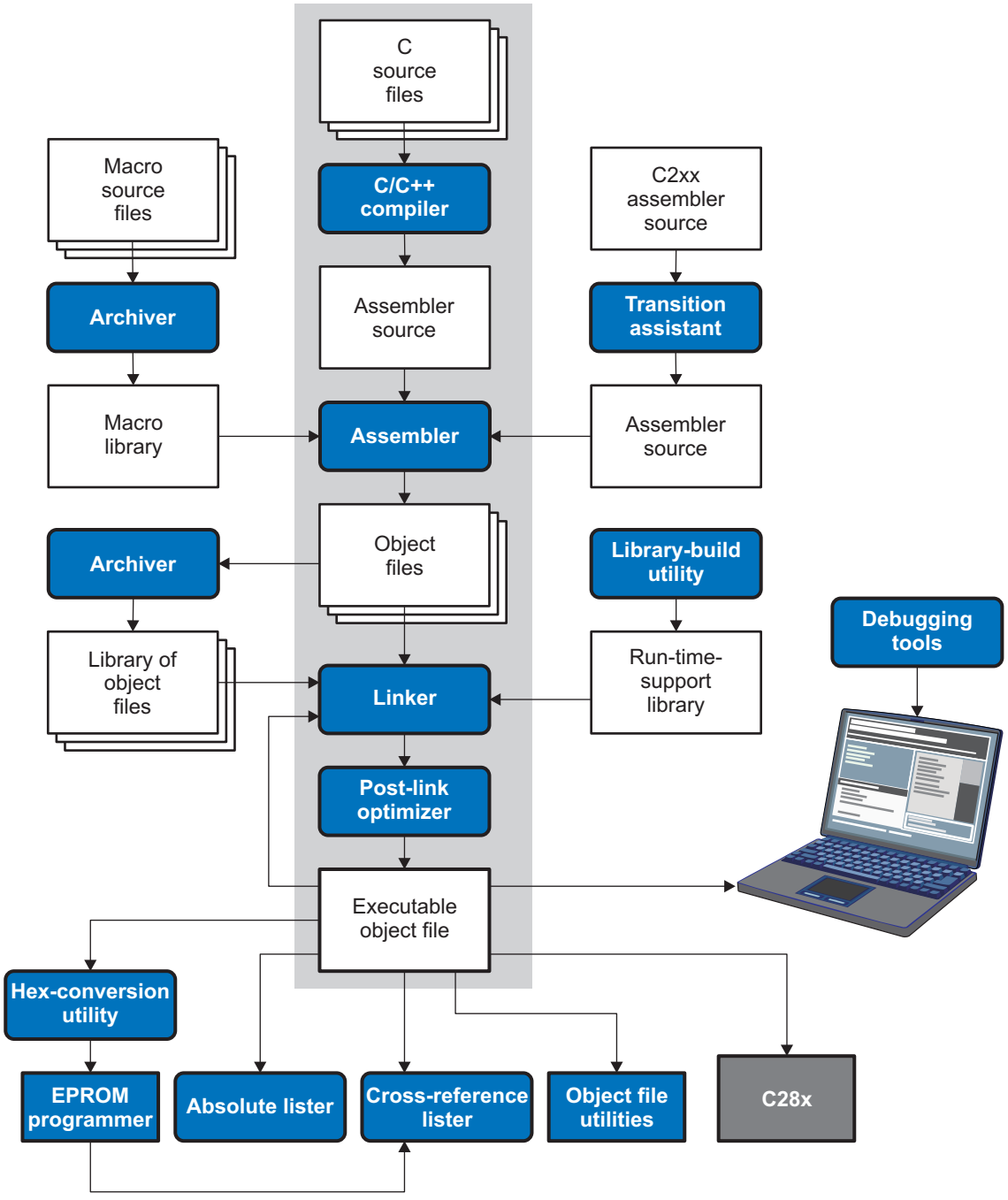
This chapter shows how these tools fit into the general software tools development flow and gives a brief description of each tool. For convenience, it also summarizes the C/C++ compiler and debugging tools. For detailed information on the compiler and debugger, and for complete descriptions of the TMS320C28x, refer to the books listed in *Related Documentation From Texas Instruments*.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1-1 shows the TMS320C28x 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. TMS320C28x 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 TMS320C28x machine code object modules. See the *TMS320C28x Optimizing C/C++ Compiler User's Guide* for more information. A **shell program**, an **optimizer**, and an **interlist utility** are included in the installation:
  - 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.
- The **assembler** translates assembly language source files into machine language object modules. Source files can contain instructions, assembler directives, and macro directives. You can use assembler directives to control the assembly process, including the source listing format, data alignment, and section content. See Chapter 4 through Chapter 6. See the *TMS320C28x DSP CPU and Instruction Set Reference Guide* for detailed information on the assembly language instruction set.
- The **linker** combines object files into a single executable object module. It performs symbolic 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 global symbols. See Chapter 8.
- The **archiver** allows you to collect a group of files into a single archive file, called a library. The most common use of the archiver is to collect a group of object files into an object library. The linker extracts object library members to resolve external references during the link. You can also use the archiver to 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 archiver allows you to modify a library by deleting, replacing, extracting, or adding members. See Section 7.1.
- 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 7.5 for more information about using the archiver to manage the content of a library.
- You can use the **library-build utility** to build your own customized run-time-support library. See the *TMS320C28x Optimizing C/C++ Compiler User's Guide* for more information.
- The **hex conversion utility** converts object files to TI-Tagged, ASCII-Hex, Intel, Motorola-S, or Tektronix object format. Converted files can be downloaded to an EPROM programmer. See Chapter 12.
- 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 9.
- 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 10.
- The main product of this development process is a executable object file that can be executed on a **TMS320C28x** device. You can use one of several debugging tools to refine and correct your code. Available products include:
  - An instruction-accurate and clock-accurate software simulator
  - An XDS emulator

In addition, the following utilities are provided to help examine or manage the content of a given object file:

- The **object file display utility** prints the contents of object files and object libraries in either human readable or XML formats. See Section 11.1.
- The **disassembler** decodes the machine code from object modules to show the assembly instructions that it represents. See Section 11.2.
- The **name utility** prints a list of symbol names for objects and functions defined or referenced in an object file or object archive. See Section 11.3.
- The **strip utility** removes symbol table and debugging information from object files and object libraries. See Section 11.4.

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 TMS320C28x 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 Object File Format Specifications

The TI Code Generation Tools for C28x support the use of either the COFF ABI or the Embedded Application Binary Interface (EABI). The ABI used is determined by the --abi command-line option. The default ABI is COFF.

The object file format generated by the tools differs depending on the ABI used:

- **COFF:** These object files conform to the Common Object File Format (COFF). See the *Common Object File Format Application Report* (SPRAAO8) for details on this format.
- **EABI:** These object files conform to the ELF (Executable and Linking Format) binary format, which is used by EABI. See the *TMS320C28x Optimizing C/C++ Compiler User's Guide* (SPRU514) for information on using the EABI ABI. The complete EABI specifications can be found in the *C28x Embedded Application Binary Interface Application Report* (SPRAC71. The ELF object files conform to the December 17, 2003 snapshot of the System V generic ABI (or gABI). This specification is currently maintained by SCO.

2.2 Executable Object Files

The linker can be used to produce static 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, have been combined and placed in 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. See Chapter 3 for details about loading and running programs.

2.3 Introduction to 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. Each section of an object file is separate and distinct.

COFF format executable object files contain *sections*.

ELF format executable object files contain *segments*. An ELF segment is a meta-section. It represents a contiguous region of target memory. It is a collection of *sections* that have the same property, such as writeable or readable. An ELF loader needs the segment information, but does not need the section information. The ELF standard allows the linker to omit ELF section information entirely from the executable object file.

Object files usually contain three default sections:

.text section	Contains executable code [1]
.data section	Usually contains initialized data
.ebss section (for COFF) **.bss section (for EABI)**	Usually reserves space for uninitialized variables

Note that the .data section is used mainly for EABI. For COFF, the compiler generates .cinit sections that are used to initialize the .ebss section. The assembler can be used to place initialized data in the .data section for both COFF and EABI.

The assembler and linker allow you to create, name, and link other kinds of sections. The .text, .data, and .ebss or .bss sections are archetypes for how sections are handled.

There are two basic types of sections:

Initialized sections	Contain data or code. The .text and .data sections are initialized; user-named sections created with the .sect assembler directive are also initialized.
Uninitialized sections	Reserve space in the memory map for uninitialized data. The .ebss or .bss section is uninitialized; user-named sections created with the .usect assembler directive are also uninitialized.

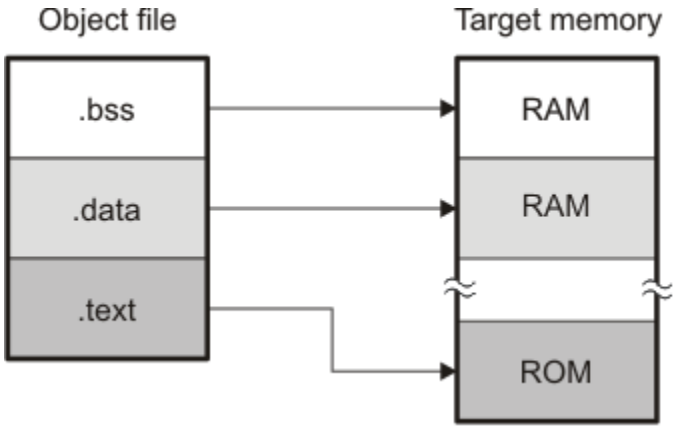
[1] Some targets allow content other than text, such as constants, in .text sections.

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 *placement*. 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 in a portion of the memory map that contains ROM. For information on section placement, see the "Specifying Where to Allocate Sections in Memory" section of the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

Figure 2-1 shows an example of the relationship between sections in an object file and a hypothetical target memory. (This figure shows COFF sections. For EABI, the .ebss section would be .bss.) ROM may be EEPROM, FLASH or some other type of physical memory in an actual system.

Figure 2-1. Partitioning Memory Into Logical Blocks



2.3.1 Special Section Names

You can use the .sect and .usect directives to create any section name you like, but certain sections are treated in a special manner by the linker and the compiler's run-time support library. If you create a section with the same name as a special section, you should take care to follow the rules for that special section.

A few common special sections are:

- .text -- Used for program code.
- .data -- Used for initialized non-const objects (global variables). (Used mainly with EABI)
- .ebss (COFF) or .bss (EABI) -- Used for uninitialized objects (global variables).
- .econst (COFF) or .const (EABI) -- Used for initialized const objects (string constants, variables declared const).
- .cinit -- Used to initialize C global variables at startup.
- .stack -- Used for the function call stack.
- .esysmem (COFF) or .sysmem (EABI) - Used for the dynamic memory allocation pool.

For more information on sections, see the "Specifying Where to Allocate Sections in Memory" section of the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

2.4 How the Assembler Handles Sections

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

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

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 2.4.6.

NOTE: If you do not use a section directive, the assembler assembles everything into the .text section.

2.4.1 Uninitialized Sections

Uninitialized sections reserve space in TMS320C28x memory; they are usually placed in 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 following assembler directives.

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

Each time you invoke the .bss or .usect directive, the assembler reserves additional space in the .bss or the user-named section. The syntax is:

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

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

size in words is an absolute expression (see Section 4.9). The .usect directive reserves *size in words* words in section name. 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 allocated space starts a page boundary. By default, the compiler causes this flag to be set to 0 so that DP load optimization is used. The compiler provides the "blocked" and "noblocked" variable attributes for controlling blocking on a per-variable basis. For examples of DP load optimization, see the Tools Insider blog in TI's E2E community.

alignment flag is an optional parameter. It causes the assembler to allocate the specified size in words on long word boundaries. The resulting alignment will be on a boundary that is 2 to the power of the specified alignment flag. For example, an alignment flag of 5 gives an alignment of 2**5, which is 32 words.

section name specifies the user-named section in which to reserve space. See Section 2.4.3.

Initialized section directives (.text, .data, and .sect) change which section is considered the *current* section (see Section 2.4.4). However, the .bss and .usect directives *do not* change the current section; they simply escape from the current section temporarily. Immediately after a .bss or .usect directive, the assembler resumes assembling into whatever the current section was before the directive. The .bss and .usect directives can appear anywhere in an initialized section without affecting its contents. For an example, see Section 2.4.7.

The .usect directive can also be used to create uninitialized subsections. See Section 2.4.6 for more information on creating subsections.

The .common directive (EABI only) is similar to directives that create uninitialized data sections, except that common symbols are created by the linker instead.

2.4.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in TMS320C28x 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 references. The following directives tell the assembler to place code or data into a section. The syntaxes for these directives are:

.text
.data
.sect "*section name***"**

The .sect directive can also be used to create initialized subsections. See Section 2.4.6, for more information on creating subsections.

2.4.3 User-Named Sections

User-named sections are sections that *you* create. You can use them like the default .text, .data, and .ebss or .bss sections, but each section with a distinct name is kept distinct during assembly.

For example, repeated use of the .text directive builds up a single .text section in the object file. This .text section is allocated in memory as a single unit. Suppose there is a portion of executable code (perhaps an initialization routine) that you want the linker to place in a different location than the rest of .text. If you assemble this segment of code into a user-named section, it is assembled separately from .text, and you can use the linker to 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 .ebss or .bss section.

These directives let you create user-named sections:

- The **.usect** directive creates uninitialized sections that are used like the .ebss or .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 user-named sections with relocatable addresses.

The syntaxes for these directives are:

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

When using COFF, you can create up to 32,767 distinct named sections. When using EABI, the maximum number of sections is 2^{32}-1 (4294967295).

The *section name* parameter is the name of the section. For the .usect and .sect directives, a section name can refer to a subsection; see Section 2.4.6 for details.

Each time you invoke one of these directives with a new name, you create a new user-named section. Each time you invoke one of these directives with a name that was already used, the assembler resumes assembling 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.

2.4.4 Current Section

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 which section is considered 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 from earlier in the file, 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 all of the code or data which was placed in that section.

2.4.5 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 the symbols in each section according to the final address of the section in which that symbol is defined. See Section 2.7 for information on relocation.

2.4.6 Subsections

A subsection is created by creating a section with a colon in its name. Subsections are logical subdivisions of larger sections. Subsections are themselves sections and can be manipulated by the assembler and linker.

The assembler has no concept of subsections; to the assembler, the colon in the name is not special. The subsection .text:rts would be considered completely unrelated to its parent section .text, and the assembler will not combine subsections with their parent sections.

Subsections are used to keep parts of a section as distinct sections so that they can be separately manipulated. For instance, by placing each function and object in a uniquely-named subsection, the linker gets a finer-grained view of the section for memory placement and unused-function elimination.

By default, when the linker sees a SECTION directive in the linker command file like ".text", it will gather .text and all subsections of .text into one large output section named ".text". You can instead use the SECTION directive to control the subsection independently. See Section 8.5.5.1 for an example.

You can create subsections in the same way you create other user-named sections: by using the .sect or .usect directive.

The syntaxes for a subsection name are:

symbol	**.usect "***section_name***:***subsection_name***",***size in words***[,***blocking flag***[,***alignment flag***]]**
	.sect "*section_name***:***subsection_name***"**

A subsection is identified by the base section name followed by a colon and the name of the subsection. The subsection name may not contain any spaces.

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.

You can create two types of subsections:

- Initialized subsections are created using the .sect directive. See Section 2.4.2.
- Uninitialized subsections are created using the .usect directive. See Section 2.4.1.

Subsections are placed in the same manner as sections. See Section 8.5.5 for information on the SECTIONS directive.

2.4.7 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 4.12 for more information on interpreting the fields in a source listing.

Figure 2-2. Using Sections Directives Example

```
 1                    ************************************************
 2                    **  Assemble an initialized table into .data.   **
 3                    ************************************************
 4  00000000                      .data
 5  00000000    0011   coeff      .word   011h, 022h, 033h
    00000001    0022
    00000002    0033
 6
 7                    ************************************************
 8                    **    Reserve space in .ebss for a variable.    **
 9                    ************************************************
10  00000000                      .bss   buffer,  10
11
12                    ************************************************
13                    **                Still in .data                **
14                    ************************************************
15  00000003    0123   ptr        .word  0123h
16
17                    ************************************************
18                    **    Assemble code into the .text section.     **
19                    ************************************************
20  00000000                      .text
21  00000000    28A1   add:       mov    ar1, #0Fh
    00000001    000F
22  00000002    0BA1   aloop:     dec    ar1
23  00000003    0009               banz   aloop,ar1--
    00000004    FFFF
24
25                    ************************************************
26                    **     Another initialized table into .data     **
27                    ************************************************
28  00000004                      .data
29  00000004    00AA   ivals      .word  0AAh,  0BBh,  0CCh
    00000005    00BB
    00000006    00CC
30
31                    ************************************************
32                    **  Define another section for more variables.  **
33                    ************************************************
34  00000000          var2        .usect "newvars", 1
35  00000001          inbuf       .usect "newvars", 7
36
37                    ************************************************
38                    **        Assemble more code into .text.        **
39                    ************************************************
40  00000005                      .text
41  00000005    28A1   end_mpy:   mov    ar1, #0Ah
    00000006    000A
42  00000007    33A1   mloop:     mpy    p,t,ar1
43  00000008    28AC               mov    t, #0Ah
    00000009    000A
44  0000000a    3FA1               mov    ar1, p
45  0000000b    6BFA               sb     end_mpy, OV
```

Field 1 Field 2 Field 3 Field 4

As Figure 2-3 shows, the example code in Figure 2-2 creates four sections:

.text	contains twelve 16-bit words of object code.
.data	contains seven 16-bit words of initialized data.
.bss	reserves ten 16-bit words in memory.
newvars	is a user-named section created with the .usect directive; it contains eight 16-bit 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 number	Object code	Section
5	0011	.data
5	0022	
5	0033	
15	0123	
29	00AA	
29	00BB	
29	00CC	
21	28A1	.text
21	000F	
22	0BA1	
23	0009	
23	FFFF	
41	28A1	
41	000A	
42	33A1	
43	28AC	
43	000A	
44	3FA1	
45	6BFB	
10	No data 10 words preserved	.bss
34	No data 8 words preserved	newvars
35		

2.5 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 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 let you manipulate the placement of sections with greater precision. You can specify the location of each subsection with the linker's SECTIONS directive. If you do not specify a subsection, the subsection is combined with the other sections with the same base section name. See Section 8.5.5.1.

It is not always necessary to use linker directives. If you do not use them, the linker uses the target processor's default placement algorithm described in Section 8.7. 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:

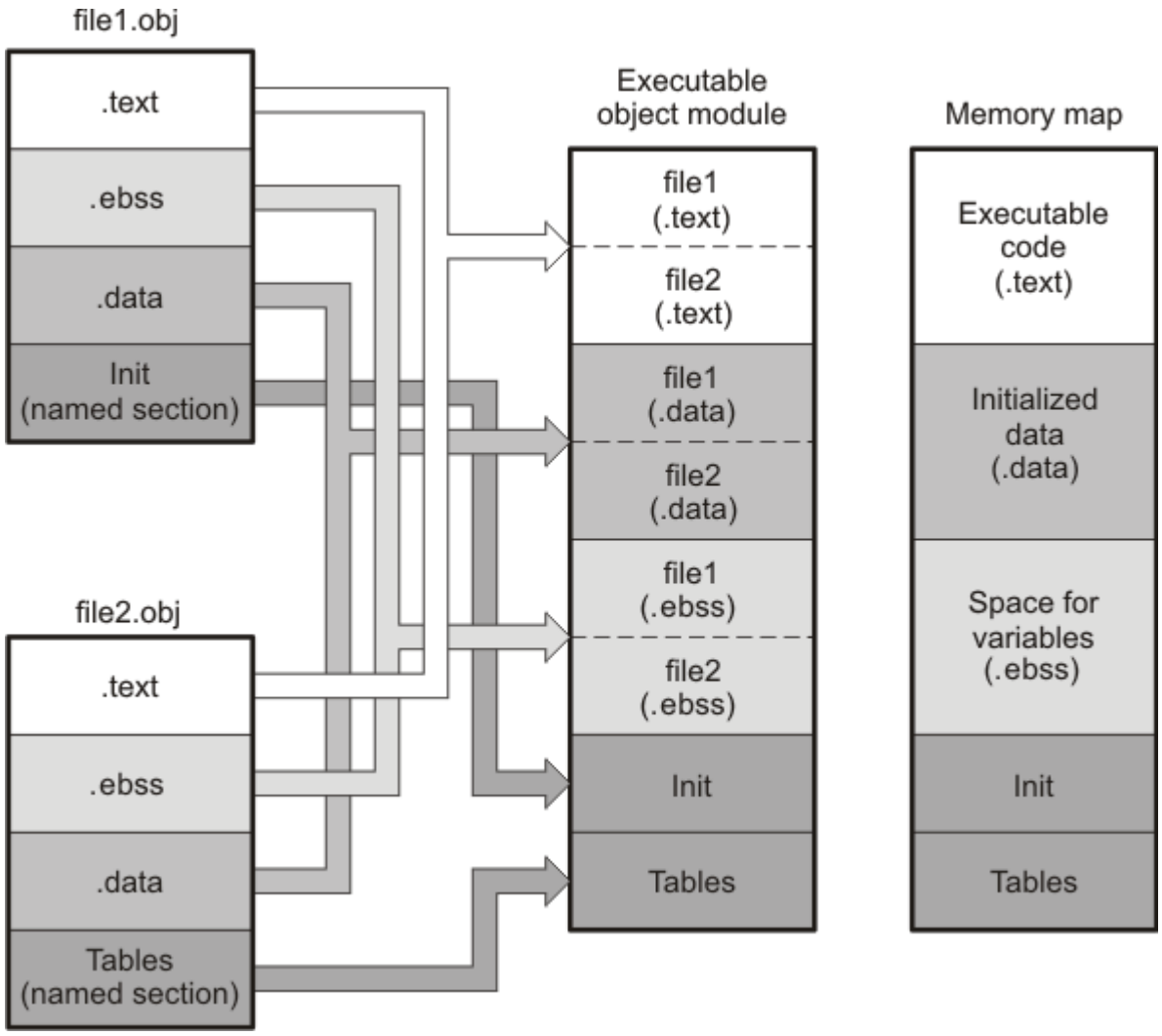
- Section 8.5, *Linker Command Files*
- Section 8.5.4, *The MEMORY Directive*
- Section 8.5.5, *The SECTIONS Directive*
- Section 8.7, *Default Placement Algorithm*

2.5.1 Combining Input Sections

Figure 2-4 provides a simplified example of the process of linking two files together.

Note that this is a simplified example, so it does not show all the sections that will be created or the actual sequence of the sections. See Section 8.7 for the actual default memory placement map for TMS320C28x. (The following figure shows sections used by COFF. For EABI, change the .ebss section to .bss.)

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 .ebss default sections; in addition, each contains a user-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 .ebss sections, and finally places the user-named sections at the end. The memory map shows the combined sections to be placed into memory.

2.5.2 Placing Sections

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 user-named section placed where the .data section would normally be allocated. Most memory maps contain various types of memory (RAM, ROM, EEPROM, FLASH, 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 8.5.4 and Section 8.5.5. See Section 8.7 for the actual default memory allocation map for TMS320C28x.

2.6 Symbols

An object file contains a symbol table that stores information about *symbols* in the object file. The linker uses this table when it performs relocation. See Section 2.7.

An object file symbol is a named 32-bit integer value, usually representing an address. A symbol can represent such things as the starting address of a function, variable, section, or an absolute integer (such as the size of the stack).

Symbols are defined in assembly by adding a label or a directive such as .set .equ .bss, or .usect.

Symbols have a *binding*, which is similar to the C concept of *linkage*. Both COFF and ELF file formats may contain symbols bound *locally* and *globally*. ELF also binds symbols as *weak symbols*.

- **Global symbols** are visible to the entire program. The linker does not allow more than one global definition of a particular symbol; it issues a multiple-definition error if a global symbol is defined more than once. (The assembler can provide a similar multiple-definition error for local symbols.) A reference to a global symbol from any object file refers to the one and only allowed global definition of that symbol. Assembly code must explicitly make a symbol global by adding a .def, .ref, or .global directive. (See Section 2.6.1.)
- **Local symbols** are visible only within one object file; each object file that uses a symbol needs its own local definition. References to local symbols in an object file are entirely unrelated to local symbols of the same name in another object file. By default, a symbol is local. (See Section 2.6.2.)
- **Weak symbols** (EABI only) are symbols that may be used but not defined in the current module. They may or may not be defined in another module. A weak symbol is intended to be overridden by a strong (non-weak) global symbol definition of the same name in another object file. If a strong definition is available, the weak symbol is replaced by the strong symbol. If no definition is available (that is, if the weak symbol is unresolved), no error is generated, but the weak variable's address is considered to be null (0). For this reason, application code that accesses a weak variable *must check that its address is not zero before attempting to access the variable*. (See Section 2.6.3.)

Absolute symbols are symbols that have a numeric value. They may be constants. To the linker, such symbols are unsigned values, but the integer may be treated as signed or unsigned depending on how it is used. The range of legal values for an absolute integer is 0 to 2^32-1 for unsigned treatment and -2^31 to 2^31-1 for signed treatment.

In general, *common symbols* (see .common directive) are preferred over weak symbols.

See Section 4.8 for information about *assembler symbols*.

2.6.1 Global (External) Symbols

Global symbols are symbols that are either accessed in the current module but defined in another (an external symbol) or defined in the current module and accessed in another. Such symbols are visible across object modules. You must use the .def, .ref, or .global directive to identify a symbol as external:

.def	The symbol is defined in the current file and may be 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 assembler chooses either .def or .ref as appropriate for each symbol.

The following code fragment illustrates these definitions.

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

x:      ADD     AR1, #56h
        B       y, UNC

q:      ADD     AR1, #56h
        B       z, UNC
```

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

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

The linker attempts to 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.

An error also occurs if the same symbol is defined more than once.

2.6.2 Local Symbols

Local symbols are visible within a single object file. Each object file may have its own local definition for a particular symbol. References to local symbols in an object file are entirely unrelated to local symbols of the same name in another object file.

By default, a symbol is local.

2.6.3 Weak Symbols

Weak symbols are symbols that may or may not be defined.

NOTE: Weak symbols are supported only in EABI mode.

The linker processes symbols that are defined with a "weak" binding differently from symbols that are defined with global binding. Instead of including a weak symbol in the object file's symbol table (as it would for a global symbol), the linker only includes a weak symbol in the output of a "final" link if the symbol is required to resolve an otherwise unresolved reference.

This allows the linker to minimize the number of symbols it includes in the output file's symbol table by omitting those that are not needed to resolve references. Reducing the size of the output file's symbol table reduces the time required to link, especially if there are a large number of pre-loaded symbols to link against. This feature is particularly helpful for OpenCL applications.

You can define a weak symbol using either the .weak assembly directive or the weak operator in the linker command file.

- **Using Assembly:** To define a weak symbol in an input object file, the source file can be written in assembly. Use the .weak and .set directives in combination as shown in the following example, which defines a weak symbol "ext_addr_sym":

```
                .weak   ext_addr_sym
ext_addr_sym    .set    0x12345678
```

  Assemble the source file that defines weak symbols, and include the resulting object file in the link. The "ext_addr_sym" in this example is available as a weak symbol in a final link. It is a candidate for removal if the symbol is not referenced elsewhere in the application. See .weak directive.

- **Using the Linker Command File:** To define a weak symbol in a linker command file, use the "weak" operator in an assignment expression to designate that the symbol as eligible for removal from the output file's symbol table if it is not referenced. In a linker command file, an assignment expression outside a MEMORY or SECTIONS directive can be used to define a weak linker-defined symbol. For example, you can define "ext_addr_sym" as follows:

```
weak(ext_addr_sym) = 0x12345678;
```

  If the linker command file is used to perform the final link, then "ext_addr_sym" is presented to the linker as a weak symbol; it will not be included in the resulting output file if the symbol is not referenced. See Section 8.6.2.

- **Using C/C++ code:** See information about the WEAK pragma and weak GCC-style variable attribute in the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

If there are multiple definitions of the same symbol, the linker uses certain rules to determine which definition takes precedence. Some definitions may have weak binding and others may have strong binding. "Strong" in this context means that the symbol has *not* been given a weak binding by either of the two methods described above. Some definitions may come from an input object file (that is, using assembly directives) and others may come from an assignment statement in a linker command file.

The linker uses the following guidelines to determine which definition is used when resolving references to a symbol:

- A strongly bound symbol always takes precedence over a weakly bound symbol.
- If two symbols are both strongly bound or both weakly bound, a symbol defined in a linker command file takes precedence over a symbol defined in an input object file.
- If two symbols are both strongly bound and both are defined in an input object file, the linker provides a symbol redefinition error and halts the link process.

2.6.4 The Symbol Table

The assembler generates entries with global (external) binding in the symbol table for each of the following:

- Each .ref, .def, or .global directive (see Section 2.6.1)
- The beginning of each section

The assembler generates entries with local binding for each locally-available function.

For informational purposes, there are also entries in the symbol table for each symbol in a program.

2.7 Symbolic Relocations

The assembler treats each section as if it began at address 0. Of course, all sections cannot actually begin at address 0 in memory, so the linker must relocate sections. For COFF, all relocations are relative to address 0 in their sections. For EABI, relocations are symbol-relative rather than section-relative.

The linker can *relocate* 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 fragment for a TMS320C28x device for which the assembler generates relocation entries.

Example 2-1. Code That Generates Relocation Entries

```
 1                        .global X
 2 00000000               .text
 3 00000000 0080'         LC      Y          ; Generates a relocation entry
   00000001 0004
 4 00000002 28A1!         MOV     AR1,#X     ; Generates a relocation entry
   00000003 0000
 5 00000004 7621  Y:      IDLE
```

2.7.1 Expressions With Multiple Relocatable Symbols (COFF Only)

Sometimes an expression contains more than one relocatable symbol, or cannot be evaluated at assembly time. In this case, the assembler encodes the entire expression in the object file. After determining the addresses of the symbols, the linker computes the value of the expression.

Expression Cannot Be Larger Than Space Reserved

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

Each section in 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, which are addresses known at assembly time). If you want the linker to retain relocation entries, invoke the linker with the --relocatable option (see Section 8.4.3.2).

In Example 2-1, both symbols X and Y are relocatable. Y is defined in the .text section of this module; X is defined in another module. When the code is assembled, X has a value of 0 (the assembler assumes all undefined external symbols have values of 0), and Y has a value of 4 (relative to address 0 in the .text section). The assembler generates two relocation entries: one for X and one for Y. The reference to X is an external reference (indicated by the **!** character in the listing). The reference to Y is to an internally defined relocatable symbol (indicated by the **'** character in the listing).

After the code is linked, suppose that X is relocated to address 0x7100. Suppose also that the .text section is relocated to begin at address 0x7200; Y now has a relocated value of 0x7204. The linker uses the two relocation entries to patch the two references in the object code:

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

Example 2-2. Simple Assembler Listing

```
1                       .global  sym1, sym2
2
3 00000000 FF20%        MOV      ACC, #(sym2-sym1)
  00000001 0000
```

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

```
00000000 FF20          MOV      ACC, #(sym2-sym1)
00000001 0100
```

2.8 Loading a Program

The linker creates an executable object file which can be loaded in several ways, depending on your execution environment. These methods include using Code Composer Studio or the hex conversion utility. For details, see Section 3.1.

Program Loading and Running

Even after a program is written, compiled, and linked into an executable object file, there are still many tasks that need to be performed before the program does its job. The program must be loaded onto the target, memory and registers must be initialized, and the program must be set to running.

Some of these tasks need to be built into the program itself. *Bootstrapping* is the process of a program performing some of its own initialization. Many of the necessary tasks are handled for you by the compiler and linker, but if you need more control over these tasks, it helps to understand how the pieces are expected to fit together.

This chapter will introduce you to the concepts involved in program loading, initialization, and startup.

This chapter does not cover *dynamic loading*.

This chapter currently provides examples for the C6000 device family. Refer to your device documentation for various device-specific aspects of bootstrapping.

Topic **Page**

3.1 Loading

A program needs to be placed into the target device's memory before it may be executed. *Loading* is the process of preparing a program for execution by initializing device memory with the program's code and data. A *loader* might be another program on the device, an external agent (for example, a debugger), or the device might initialize itself after power-on, which is known as *bootstrap loading*, or *bootloading*.

The loader is responsible for constructing the *load image* in memory before the program starts. The load image is the program's code and data in memory before execution. What exactly constitutes loading depends on the environment, such as whether an operating system is present. This section describes several loading schemes for bare-metal devices. This section is not exhaustive. Additionally, with the COFF RAM model, the loader is responsible for parsing the .cinit section and performing the initializations encoded therein at load time.

A program may be loaded in the following ways:

- **A debugger running on a connected host workstation.** In a typical embedded development setup, the device is subordinate to a host running a debugger such as Code Composer Studio (CCS). The device is connected with a communication channel such as a JTAG interface. CCS reads the program and writes the load image directly to target memory through the communications interface.
- **"Burning" the load image onto an EPROM module.** The hex converter (hex2000) can assist with this by converting the executable object file into a format suitable for input to an EPROM programmer. The EPROM is placed onto the device itself and becomes a part of the device's memory. See Chapter 12 for details.
- **Bootstrap loading from a dedicated peripheral, such as an I^2C peripheral.** The device may require a small program called a bootloader to perform the loading from the peripheral. The hex converter can assist in creating a bootloader.
- **Another program running on the device.** The running program can create the load image and transfer control to the loaded program. If an operating system is present, it may have the ability to load and run programs.

3.1.1 Load and Run Addresses

Consider an embedded device for which the program's load image is burned onto EPROM/ROM. Variable data in the program must be writable, and so must be located in writable memory, typically RAM. However, RAM is *volatile*, meaning it will lose its contents when the power goes out. If this data must have an initial value, that initial value must be stored somewhere else in the load image, or it would be lost when power is cycled. The initial value must be copied from the non-volatile ROM to its run-time location in RAM before it is used. See Section 8.8 for ways this is done.

The *load address* is the location of an object in the load image.

The *run address* is the location of the object as it exists during program execution.

An *object* is a chunk of memory. It represents a section, segment, function, or data.

The load and run addresses for an object may be the same. This is commonly the case for program code and read-only data, such as the .econst section. In this case, the program can read the data directly from the load address. Sections that have no initial value, such as the .ebss section, do not have load data and are considered to have load and run addresses that are the same. If you specify different load and run addresses for an uninitialized section, the linker provides a warning and ignores the load address.

The load and run addresses for an object may be different. This is commonly the case for writable data, such as the .data section. The .data section's starting contents are placed in ROM and copied to RAM. This often occurs during program startup, but depending on the needs of the object, it may be deferred to sometime later in the program as described in Section 3.5.

Symbols in assembly code and object files almost always refer to the run address. When you look at an address in the program, you are almost always looking at the run address. The load address is rarely used for anything but initialization.

The load and run addresses for a section are controlled by the linker command file and are recorded in the object file metadata.

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 examples that specify load and run addresses, see Section 8.5.6.1.

For an example that illustrates how to move a block of code at run time, see Example 8-10. To create a symbol that lets you refer to the load-time address, rather than the run-time address, see the .label directive. To use copy tables to copy objects from load-space to run-space at boot time, see Section 8.8.

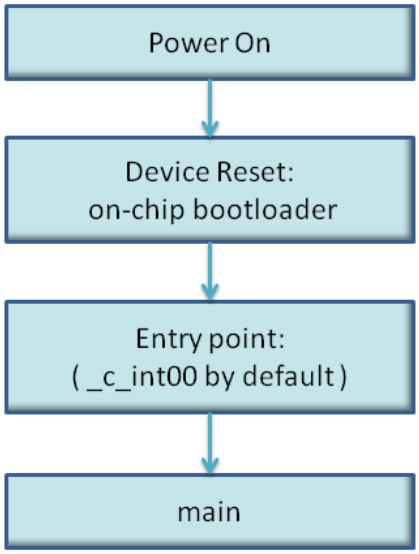
ELF format executable object files contain *segments*. See Section 2.3 for information about sections and segments. COFF format executable object files contain *sections*.

3.1.2 Bootstrap Loading

The details of bootstrap loading (bootloading) vary a great deal between devices. Not every device supports every bootloading mode, and using the bootloader is optional. This section discusses various bootloading schemes to help you understand how they work. Refer to your device's data sheet to see which bootloading schemes are available and how to use them.

A typical embedded system uses bootloading to initialize the device. The program code and data may be stored in ROM or FLASH memory. At power-on, an on-chip bootloader (the *primary bootloader*) built into the device hardware starts automatically.

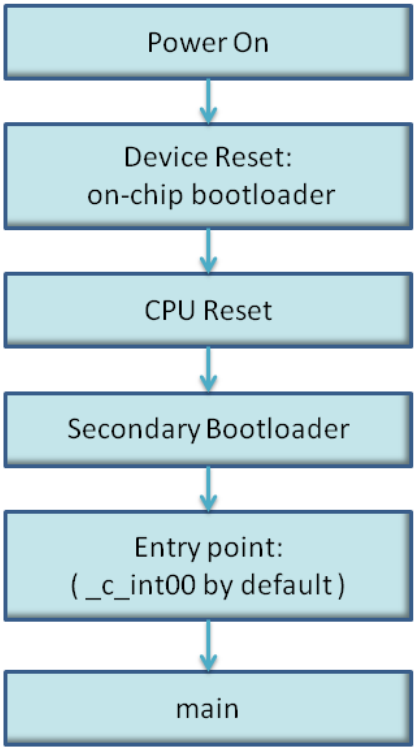
Figure 3-1. Bootloading Sequence (Simplified)



The primary bootloader is typically very small and copies a limited amount of memory from a dedicated location in ROM to a dedicated location in RAM. (Some bootloaders support copying the program from an I/O peripheral.) After the copy is completed, it transfers control to the program.

For many programs, the primary bootloader is not capable of loading the entire program, so these programs supply a more capable secondary bootloader. The primary bootloader loads the secondary bootloader and transfers control to it. Then, the secondary bootloader loads the rest of the program and transfers control to it. There can be any number of layers of bootloaders, each loading a more capable bootloader to which it transfers control.

Figure 3-2. Bootloading Sequence with Secondary Bootloader



3.1.2.1 Boot, Load, and Run Addresses

The *boot address* of a bootloaded object is where its raw data exists in ROM before power-on.

The boot, load, and run addresses for an object may all be the same; this is commonly the case for .const data. If they are different, the object's contents must be copied to the correct location before the object may be used.

The boot address may be different than the load address. The bootloader is responsible for copying the raw data to the load address.

The boot address is not controlled by the linker command file or recorded in the object file; it is strictly a convention shared by the bootloader and the program.

3.1.2.2 Primary Bootloader

The detailed operation of the primary bootloader is device-specific. Some devices have complex capabilities such as booting from an I/O peripheral or configuring memory controller parameters.

3.1.2.3 Secondary Bootloader

The hex converter assumes the secondary bootloader is of a particular format. The hex converter's model bootloader uses a *boot table*. You can use whatever format you want, but if you follow this model, the hex converter can create the boot table automatically.

3.1.2.4 Boot Table

The input for the model secondary bootloader is the *boot table*. The boot table contains records that instruct the secondary bootloader to copy blocks of data contained in the table to specified destination addresses. The hex conversion utility automatically builds the boot table for the secondary bootloader. Using the utility, you specify the sections you want to initialize, the boot table location, and the name of the section containing the secondary bootloader routine and where it should be located. The hex conversion utility builds a complete image of the table and adds it to the program.

The boot table is target-specific. For C6000, the format of the boot table is simple. A header record contains a 4-byte field that indicates where the boot loader should branch after it has completed copying data. After the header, each section that is to be included in the boot table has the following contents:

- 4-byte field containing the size of the section
- 4-byte field containing the destination address for the copy
- the raw data
- 0 to 3 bytes of trailing padding to make the next field aligned to 4 bytes

More than one section can be entered; a termination block containing an all-zero 4-byte field follows the last section.

See Section 12.11.2 for details about the boot table format.

3.1.2.5 Bootloader Routine

The bootloader routine is a normal function, except that it executes before the C environment is set up. For this reason, it can't use the C stack, and it can't call any functions that have yet to be loaded!

The following sample code is for C6000 and is from *Creating a Second-Level Bootloader for FLASH Bootloading on TMS320C6000 Platform With Code Composer Studio* (SPRA999).

Example 3-1. Sample Secondary Bootloader Routine

```
;  ======== boot_c671x.s62 ========

; global EMIF symbols defined for the c671x family
            .include        boot_c671x.h62
            .sect ".boot_load"
            .global _boot
_boot:
;*************************************************************************
;* DEBUG LOOP - COMMENT OUT B FOR NORMAL OPERATION
;*************************************************************************
zero B1
_myloop: ;  [!B1] B _myloop
            nop  5
_myloopend: nop
;*************************************************************************
;* CONFIGURE EMIF
;*************************************************************************
        ;***************************************************************
        ; *EMIF_GCTL = EMIF_GCTL_V;
        ;***************************************************************
            mvkl  EMIF_GCTL,A4
      ||    mvkl  EMIF_GCTL_V,B4
            mvkh  EMIF_GCTL,A4
      ||    mvkh  EMIF_GCTL_V,B4
            stw   B4,*A4
        ;***************************************************************
        ; *EMIF_CE0 = EMIF_CE0_V
        ;***************************************************************
            mvkl  EMIF_CE0,A4
      ||    mvkl  EMIF_CE0_V,B4
            mvkh  EMIF_CE0,A4
      ||    mvkh  EMIF_CE0_V,B4
```

Example 3-1. Sample Secondary Bootloader Routine (continued)

```
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_CE1 = EMIF_CE1_V (setup for 8-bit async)
        ;*****************************************************************
            mvkl  EMIF_CE1,A4
      ||    mvkl  EMIF_CE1_V,B4
            mvkh  EMIF_CE1,A4
      ||    mvkh  EMIF_CE1_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_CE2 = EMIF_CE2_V (setup for 32-bit async)
        ;*****************************************************************
            mvkl  EMIF_CE2,A4
      ||    mvkl  EMIF_CE2_V,B4
            mvkh  EMIF_CE2,A4
      ||    mvkh  EMIF_CE2_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_CE3 = EMIF_CE3_V (setup for 32-bit async)
        ;*****************************************************************
      ||    mvkl  EMIF_CE3,A4
      ||    mvkl  EMIF_CE3_V,B4      ;
            mvkh  EMIF_CE3,A4
      ||    mvkh  EMIF_CE3_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_SDRAMCTL = EMIF_SDRAMCTL_V
        ;*****************************************************************
      ||    mvkl  EMIF_SDRAMCTL,A4
      ||    mvkl  EMIF_SDRAMCTL_V,B4      ;
            mvkh  EMIF_SDRAMCTL,A4
      ||    mvkh  EMIF_SDRAMCTL_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_SDRAMTIM = EMIF_SDRAMTIM_V
        ;*****************************************************************
      ||    mvkl  EMIF_SDRAMTIM,A4
      ||    mvkl  EMIF_SDRAMTIM_V,B4      ;
            mvkh  EMIF_SDRAMTIM,A4
      ||    mvkh  EMIF_SDRAMTIM_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_SDRAMEXT = EMIF_SDRAMEXT_V
        ;*****************************************************************
      ||    mvkl  EMIF_SDRAMEXT,A4
      ||    mvkl  EMIF_SDRAMEXT_V,B4      ;
            mvkh  EMIF_SDRAMEXT,A4
      ||    mvkh  EMIF_SDRAMEXT_V,B4
            stw   B4,*A4
;*****************************************************************************
; copy sections
;*****************************************************************************
            mvkl  COPY_TABLE, a3 ; load table pointer
            mvkh  COPY_TABLE, a3
            ldw   *a3++, b1      ; Load entry point
copy_section_top:
            ldw   *a3++, b0      ; byte count
            ldw   *a3++, a4      ; ram start address
            nop   3
[!b0]       b copy_done          ; have we copied all sections?
            nop   5
copy_loop:
            ldb   *a3++,b5
            sub   b0,1,b0        ; decrement counter
```

Example 3-1. Sample Secondary Bootloader Routine (continued)

```
[ b0]       b     copy_loop      ; setup branch if not done
[!b0]       b     copy_section_top
            zero  a1
[!b0]       and   3,a3,a1
            stb   b5,*a4++
[!b0]       and   -4,a3,a5       ; round address up to next multiple of 4
[ a1]       add   4,a5,a3        ; round address up to next multiple of 4
;*****************************************************************************
; jump to entry point
;*****************************************************************************
copy_done:
            b    .S2 b1
            nop  5
```

3.2 Entry Point

The entry point is the address at which the execution of the program begins. This is the address of the startup routine. The startup routine is responsible for initializing and calling the rest of the program. For a C/C++ program, the startup routine is usually named _c_int00 (see Section 3.3.1). After the program is loaded, the value of the entry point is placed in the PC register and the CPU is allowed to run.

The object file has an entry point field. For a C/C++ program, the linker will fill in _c_int00 by default. You can select a custom entry point; see Section 8.4.13. The device itself cannot read the entry point field from the object file, so it has to be encoded in the program somewhere.

- If you are using a bootloader, the boot table includes an entry point field. When it finishes running, the bootloader branches to the entry point.
- If you are using an interrupt vector, the entry point is installed as the RESET interrupt handler. When RESET is applied, the startup routine will be invoked.
- If you are using a hosted debugger, such as CCS, the debugger may explicitly set the program counter (PC) to the value of the entry point.

3.3 Run-Time Initialization

After the load image is in place, the program can run. The subsections that follow describe bootstrap initialization of a C/C++ program. An assembly-only program may not need to perform all of these steps.

3.3.1 The _c_int00 Function

The function _c_int00 is the *startup routine* (also called the *boot routine*) for C/C++ programs. It performs all the steps necessary for a C/C++ program to initialize itself.

The name _c_int00 means that it is the interrupt handler for interrupt number 0, RESET, and that it sets up the C environment. Its name need not be exactly _c_int00, but the linker sets _c_int00 as the entry point for C programs by default. The compiler's run-time-support library provides a default implementation of _c_int00.

The startup routine is responsible for performing the following actions:

1. Set up status and configuration registers
2. Set up the stack
3. Process the .cinit run-time initialization table to autoinitialize global variables (when using the --rom_model option)
4. Call all global object constructors in .init_array (for EABI) .pinit (for COFF)
5. Call the function main
6. Call exit when main returns

3.3.2 RAM Model vs. ROM Model

Choose a startup model based on the needs of your application. The ROM model performs more work during the boot routine. The RAM model performs more work while loading the application.

If your application is likely to need frequent RESETs or is a standalone application, the ROM model may be a better choice, because the boot routine will have all the data it needs to initialize RAM variables. However, for a system with an operating system, it may be better to use the RAM model.

In the COFF RAM model, the loader is first responsible for processing the .cinit section. The .cinit section is a NOLOAD section, which means it does not get allocated to target memory. Instead, the loader is responsible for parsing the .cinit section and performing the initializations encoded therein at load time.

In both the COFF ROM and EABI ROM models, the C boot routine copies data from the .cinit section to the run-time location of the variables to be initialized.

In the EABI RAM model, no .cinit records are generated at startup.

3.3.2.1 Autoinitializing Variables at Run Time (--rom_model)

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

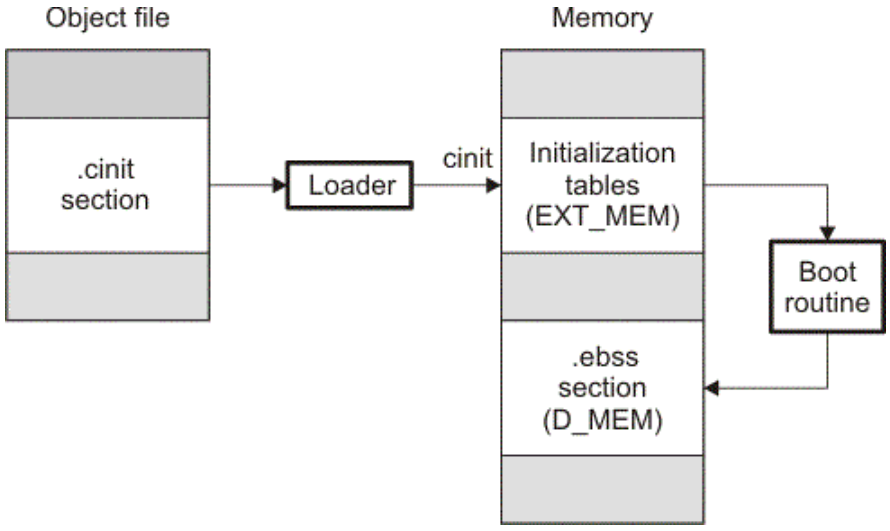
The ROM model allows initialization data to be stored in slow non-volatile memory and copied to fast memory each time the program is reset. Use this method if your application runs from code burned into slow memory or needs to survive a reset.

For the ROM model with EABI, the .cinit section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called __TI_CINIT_Base 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 run-time location of the variables.

For the ROM model with COFF, 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 .ebss or user-defined section.

Figure 3-3 illustrates autoinitialization at run time for the COFF ABI using the ROM model.

Figure 3-3. Autoinitialization at Run Time



3.3.2.2 Initializing Variables at Load Time (--ram_model)

The RAM model Initializes variables at load time. To use this method, invoke the linker with the --ram_model option.

This model may reduce boot time and save memory used by the initialization tables.

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.)

For COFF, 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.

For EABI, the linker sets __TI_CINIT_Base equal to __TI_CINIT_Limit to indicate there are no .cinit records.

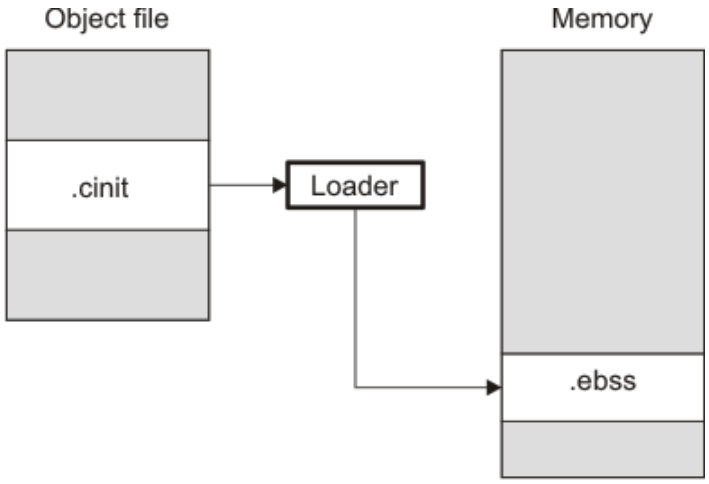
A COFF loader must be able to perform the following tasks to use initialization at load time:

- 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.

For EABI, the loader copies values directly from the .data section to memory.

Figure 3-4 illustrates the initialization of variables at load time for the COFF ABI..

Figure 3-4. Initialization at Load Time



3.3.2.3 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.c.obj. Referencing _c_int00 ensures that boot.c.obj is automatically linked in from the appropriate run-time-support library.
- For COFF, the .cinit output section is padded with a termination record to tell the boot routine (autoinitialize at run time) or the loader (initialize at load time) when to stop reading initialization tables.
- When you use the RAM model to initialize at load time (--ram_model option):
  - For COFF, the linker sets cinit to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - For EABI, the linker defines a special symbol called __TI_CINIT_Base 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 run-time location of the variables.
  - For COFF, the linker sets the STYP_COPY flag (0010h) in the .cinit section header. STYP_COPY is a 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 use the ROM model to autoinitialize at run time (--rom_model option):

- For COFF, 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.
- For EABI, the linker sets __TI_CINIT_Base equal to __TI_CINIT_Limit to indicate there are no .cinit records.

Loader

NOTE: A loader is not included as part of the TMS320C28x C/C++ compiler tools. Use Code Composer Studio as a loader.

3.3.3 About Linker-Generated Copy Tables

The RTS function copy_in can be used at run-time to move code and data around, usually from its load address to its run address. This function reads size and location information from copy tables. The linker automatically generates several kinds of copy tables. Refer to Section 8.8.

You can create and control code overlays with copy tables. See Section 8.8.4 for details and examples.

Copy tables can be used by the linker to implement run-time relocations as described in Section 3.5, however copy tables require a specific table format.

3.3.3.1 BINIT

The BINIT (boot-time initialization) copy table is special in that the target will automatically perform the copying at auto-initialization time. Refer to Section 8.8.4.2 for more about the BINIT copy table name. The BINIT copy table is copied before .cinit processing.

3.3.3.2 CINIT

EABI .cinit tables are special kinds of copy tables. Refer to Section 3.3.2.1 for more about using the .cinit section with the ROM model and Section 3.3.2.2 for more using it with the RAM model.

COFF .cinit tables can be used to provide copy table functionality. See Section 8.8 for more information.

3.4 Arguments to main

Some programs expect arguments to main (argc, argv) to be valid. Normally this isn't possible for an embedded program, but the TI runtime does provide a way to do it. The user must allocate an .args section of an appropriate size using the --args linker option. It is the responsibility of the loader to populate the .args section. It is not specified how the loader determines which arguments to pass to the target. The format of the arguments is the same as an array of pointers to char on the target.

See Section 8.4.4 for information about allocating memory for argument passing.

3.5 Run-Time Relocation

At times you may want to load code into one area of memory and move it to another area before running it. 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. Because internal memory is limited, you might swap in different speed-critical functions at different times.

The linker provides a 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. See Section 3.1.1 for more about load and run addresses. If a section is assigned two addresses at link time, 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.

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. The two sections are the same size if the load section is not compressed.

Uninitialized sections (such as .ebss or .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 8.5.6.

3.6 Additional Information

See the following sections and documents for additional information:

Section 8.4.4, "Allocate Memory for Use by the Loader to Pass Arguments (--arg_size Option)"

Section 8.4.13, "Define an Entry Point (--entry_point Option)"

Section 8.5.6.1 ,"Specifying Load and Run Addresses"

Section 8.8, "Linker-Generated Copy Tables"

Section 8.11.1, "Run-Time Initialization"

Chapter 12, "Hex Conversion Utility Description"

"Run-Time Initialization" and "System Initialization" sections in the *TMS320C28x Optimizing C/C++ Compiler User's Guide*

Assembler Description

The TMS320C28x assembler translates assembly language source files into machine language object files. These files are object modules, which are discussed in Chapter 2. Source files can contain the following assembly language elements:

Assembler directives	described in Chapter 5
Macro directives	described in Chapter 6
Assembly language instructions	described in the *TMS320C28x DSP CPU and Instruction Set Reference Guide*.

Topic **Page**

4.1 Assembler Overview

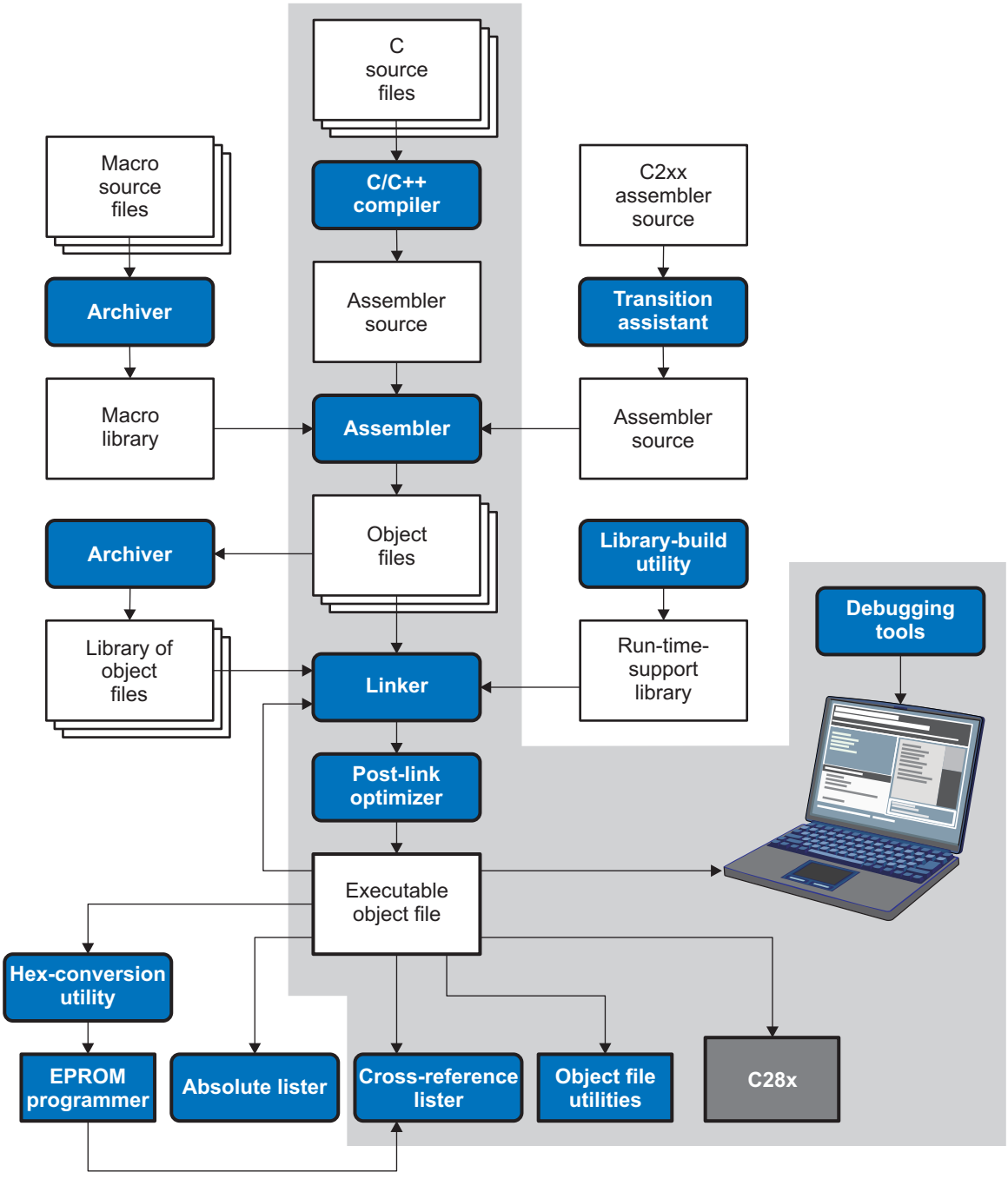
The 2-pass assembler does the following:

- Processes the source statements in a text file to produce a relocatable object file
- Produces a source listing (if requested) and provides you with control over this listing
- Allows you to 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

4.2 The Assembler's Role in the Software Development Flow

Figure 4-1 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 TMS320C28x C/C++ compiler.

Figure 4-1. The Assembler in the TMS320C28x Software Development Flow



4.3 Invoking the Assembler

To invoke the assembler, enter the following:

cl2000 *input file* [*options*]

cl2000 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 invokes 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 4-1.

Table 4-1. TMS320C28x 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.
--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 4.8.5.
--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_listing_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 --asm_listing_cross_reference option, the assembler creates a listing file automatically, naming it with the same name as the input file with a .lst extension.
--asm_undefine=*name*	**-au**	Undefines the predefined constant *name*, which overrides any --asm_define options for the specified constant.
--cla_support[=cla0|cla1|cla2]		Specifies TMS320C28x Control Law Accelerator (CLA) Type 0, Type 1, or Type 2 support. This option is used to compile or assemble code written for the CLA. This option does not need any special library support when linking; the libraries used for C28x with/without FPU support should be sufficient.
--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"
--float_support={ fpu32 | fpu64 }		Assembles code for C28x with 32-bit or 64-bit hardware FPU support.
--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 4.5.1.
--quiet	**-q**	Suppresses the banner and progress information (assembler runs in quiet mode).
--symdebug:dwarf or **--symdebug:none**	**-g**	(DWARF is on by default) 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 this option on assembly code that contains .line directives. See Section 4.13.

Table 4-1. TMS320C28x Assembler Options (continued)

Option	Alias	Description
--vcu_support[=vcu0|vcu2|vcrc]		Specifies support for the Viterbi, Complex Math and CRC Unit (VCU), Type 0 or Type 2. The default is vcu0. Note that there is no VCU Type 1. This option is useful only if the source is in assembly code, written for the VCU. The option is ignored for C/C++ code. This option does not require any special library support from the linker; the libraries used for C28x with/without VCU support should be sufficient.

4.4 Controlling Application Binary Interface

An Application Binary Interface (ABI) defines the low level interface between object files, and between an executable and its execution environment. The ABI exists to allow ABI-compliant object code to link together, regardless of its source, and allows the resulting executable to run on any system that supports that ABI.

The C28x Code Generation Tools support both the COFF ABI and the EABI ABI. The ABI used by the assembler is determined by the --abi command-line option.

- **COFF ABI:** (--abi=coffabi, the default)

  When using the COFF ABI, the output files are in COFF format. See the *Common Object File Format Application Report* (SPRAAO8) for details.
- **EABI:** (--abi=eabi)

  When using EABI, the output files are in Executable and Linking Format (ELF). See the *TMS320C28x Optimizing C/C++ Compiler User's Guide* (SPRU514) and the *C28x Embedded Application Binary Interface Application Report* (SPRAC71) for information on the EABI ABI.

All object files in an application must be built for the same ABI. The linker detects situations where object modules conform to different ABIs and generates an error.

Converting an assembly file from the COFF API to EABI requires some changes to the assembly code. For example, you would typically need to make the following changes when a memory section is referenced:

- .ebss to .bss
- .esysmem to .sysmem
- .econst to .const
- .pinit to .init_array

The examples in this guide are generally written for the COFF ABI.

4.5 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 5 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 C2000_A_DIR environment variable
4. Any directories named with the C2000_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 4.5.1) or the C2000_A_DIR environment variable (described in Section 4.5.2). The C2000_C_DIR environment variable is discussed in the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

4.5.1 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:

cl2000 --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)	`cl2000 --include_path=/tools/files source.asm`
Windows	`cl2000 --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.

4.5.2 Using the C2000_A_DIR Environment Variable

An environment variable is a system symbol that you define and assign a string to. The assembler uses the C2000_A_DIR environment variable to name alternate directories that contain copy/include files or macro libraries.

The assembler looks for the C2000_A_DIR environment variable and then reads and processes it. If the assembler does not find the C2000_A_DIR variable, it then searches for C2000_C_DIR. The processor-specific variables are useful when you are using Texas Instruments tools for different processors at the same time.

See the *TMS320C28x Optimizing C/C++ Compiler User's Guide* for details on C2000_C_DIR.

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

Operating System	Enter
UNIX (Bourne Shell)	**C2000_A_DIR="** *pathname*$_1$ **;** *pathname*$_2$ **; . . . "; export C2000_A_DIR**
Windows	**set C2000_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 C28X_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 C28X_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)	`C2000_A_DIR="/dsys"; export C2000_A_DIR` `cl2000 --include_path=/tools/files source.asm`
Windows	`C2000_A_DIR=c:\dsys` `cl2000 --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 C2000_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 C2000_A_DIR`
Windows	`set C2000_A_DIR=`

4.6 Source Statement Format

Each line in a TMS320C28x assembly input file can be empty, a comment, an assembler directive, a macro invocation, or an assembly instruction.

Assembly language source statements can contain four ordered fields (label, mnemonic, operand list, and comment). The general syntax for source statements is as follows:

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

Labels cannot be placed on instructions that have parallel bars.

Following are examples of source statements:

```
two     .set  2           ; Symbol two = 2
Begin:  MOV   AR1,#two    ; Load AR1 with 2
        .word 016h        ; Initialize a word with 016h
```

The C28x assembler reads an unlimited number of characters per line. Source statements that extend beyond 400 characters in length (including comments) are truncated in the listing file.

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional for most statements; 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.

NOTE: A mnemonic cannot begin in column 1 or it will be interpreted as a label. Mnemonic opcodes and assembler directive names without the . prefix are valid label names. Remember to always use whitespace before the mnemonic, or the assembler will think the identifier is a new label definition.

The following sections describe each of the fields.

4.6.1 Label Field

A label must be a legal identifier (see Section 4.8.1) placed in column 1. Every instruction may optionally have a label. Many directives allow a label, and some require a label.

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 on an assembly instruction or data directive, an *assembler symbol* (Section 4.8) with the same name is created. Its value is the current value of the *section program counter* (SPC, see Section 2.4.5). This symbol represents the address of that instruction. In the following example, the .word directive is used to create an array of 3 words. Because a label was used, the assembly symbol Start refers to the first word, and the symbol will have the value 40h.

```
.    .    .    .
       9                    * Assume some code was assembled
      10 000040 000A  Start:  .word 0Ah,3,7
         000044 0003
         000048 0007
```

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

```
       1 000000          Here:
       2 000000 0003         .word 3
```

A label on a line by itself is equivalent to writing:

```
Here: .equ $  ; $ provides the current value of the SPC
```

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

4.6.2 Mnemonic Field

The mnemonic field follows the label field. The mnemonic field cannot start in column 1; if it does, it is interpreted as a label. The mnemonic field can begin with pipe symbols (||) when the previous instruction is a RPT. Pipe symbols that follow a RPT instruction indicate instructions that are repeated. For example:

```
    RPT
||  Inst2 <------ This instruction is repeated.
```

In the case of C28x with FPU support, the mnemonic field can begin with pipe symbols to indicate instructions that are to be executed in parallel. For example, in the instance given below, Inst1 and Inst2 are FPU instructions that execute in parallel:

```
    Instr1
||  Instr2
```

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

- Machine-instruction mnemonic (such as ADD, MOV, or B)
- Assembler directive (such as .data, .list, .equ)
- Macro directive (such as .macro, .var, .mexit)
- Macro invocation

4.6.3 Operand Field

The operand field follows the mnemonic field and contains zero or more comma-separated operands. An operand can be one of the following:

- an immediate operand (usually a constant or symbol) (see Section 4.7 and Section 4.8)
- a register operand
- a memory reference operand
- an expression that evaluates to one of the above (see Section 4.9)

An *immediate operand* is encoded directly in the instruction. The value of an immediate operand must be a *constant expression*. Most instructions with an immediate operand require an *absolute constant expression*, such as 1234. Some instructions (such as a call instruction) allow a *relocatable constant expression*, such as a symbol defined in another file. (See Section 4.9 for details about types of expressions.)

A *register operand* is a special pre-defined symbol that represents a CPU register.

A *memory reference operand* uses one of several memory addressing modes to refer to a location in memory. Memory reference operands use a special target-specific syntax defined in the appropriate *CPU and Instruction Set Reference Guide*.

You must separate operands with commas. Not all operand types are supported for all operands. See the description of the specific instruction in the *CPU and Instruction Set Reference Guide* for your device family.

4.6.4 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.

4.7 Literal Constants

A *literal constant* (also known as a *literal* or in some other documents as an *immediate value*) is a value that represents itself, such as 12, 3.14, or "hello".

The assembler supports several types of literals:

- Binary integer literals
- Octal integer literals
- Decimal integer literals
- Hexadecimal integer literals
- Character literals
- Character string literals
- Floating-point literals

Error checking for invalid or incomplete literals is performed.

4.7.1 Integer Literals

The assembler maintains each integer literal internally as a 32-bit signless quantity. Literals are considered unsigned values, and are not sign extended. For example, the literal 00FFh is equal to 00FF (base 16) or 255 (base 10); it *does not* equal -1. which is 0FFFFFFFFh (base 16). Note that if you store 0FFh in a .byte location, the bits will be exactly the same as if you had stored -1. It is up to the reader of that location to interpret the signedness of the bits.

4.7.1.1 Binary Integer Literals

A binary integer literal is a string of up to 32 binary digits (0s and 1s) followed by the suffix B (or b). Binary literals of the form "0[bB][10]+" are also supported. 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 literals:

00000000B	Literal equal to 0_{10} or 0_{16}
0100000b	Literal equal to 32_{10} or 20_{16}
01b	Literal equal to 1_{10} or 1_{16}
11111000B	Literal equal to 248_{10} or $0F8_{16}$
0b00101010	Literal equal to 42_{10} or $2A_{16}$
0B101010	Literal equal to 42_{10} or $2A_{16}$

4.7.1.2 Octal Integer Literals

An octal integer literal is a string of up to 11 octal digits (0 through 7) followed by the suffix Q (or q). Octal literals may also begin with a 0, contain no 8 or 9 digits, and end with no suffix. These are examples of valid octal literals:

10Q	Literal equal to 8_{10} or 8_{16}
054321	Literal equal to 22737_{10} or $58D1_{16}$
100000Q	Literal equal to 32768_{10} or 8000_{16}
226q	Literal equal to 150_{10} or 96_{16}

4.7.1.3 Decimal Integer Literals

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

1000	Literal equal to 1000_{10} or $3E8_{16}$
-32768	Literal equal to $-32\ 768_{10}$ or -8000_{16}
25	Literal equal to 25_{10} or 19_{16}
4815162342	Literal equal to 4815162342_{10} or $11F018BE6_{16}$

4.7.1.4 Hexadecimal Integer Literals

A hexadecimal integer literal is a string of up to eight hexadecimal digits followed by the suffix H (or h) or preceded by 0x. *A hexadecimal literal must begin with a decimal value (0-9) if it is indicated by the H or h suffix.*

Hexadecimal digits include the decimal values 0-9 and the letters A-F or a-f. If fewer than eight hexadecimal digits are specified, the assembler right-justifies the bits.

These are examples of valid hexadecimal literals:

78h	Literal equal to 120_{10} or 0078_{16}
0x78	Literal equal to 120_{10} or 0078_{16}
0Fh	Literal equal to 15_{10} or $000F_{16}$
37ACh	Literal equal to 14252_{10} or $37AC_{16}$

4.7.1.5 Character Literals

A character literal 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 literal. A character literal consisting only of two single quotes is valid and is assigned the value 0. These are examples of valid character literals:

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

Notice the difference between character *literals* and character *string literals* (Section 4.7.2 discusses character strings). A character literal represents a single integer value; a string is a sequence of characters.

4.7.2 Character String Literals

A character string is a sequence 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

4.7.3 Floating-Point Literals

A floating-point literal is a string of decimal digits followed by a required decimal point, an optional fractional portion, and an optional 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 floating-point literals:

```
3.0
3.14
3.
-0.314e13
+314.59e-2
```

The assembler syntax does not support all C89-style float literals nor C99-style hexadecimal constants, but the $strtod built-in mathematical function supports both. If you want to specify a floating-point literal using one of those formats, use $strtod. For example:

```
$strtod(".3")
$strtod("0x1.234p-5")
```

You cannot directly use NaN, Inf, or -Inf as floating-point literals. Instead, use $strtod to express these values. The "NaN" and "Inf" strings are handled case-insensitively.

```
$strtod("NaN")
$strtod("Inf")
```

4.8 Assembler Symbols

An assembler symbol is a named 32-bit signless integer value, usually representing an address or absolute integer. A symbol can represent such things as the starting address of a function, variable, or section. The name of a symbol must be a legal identifier. The identifier becomes a symbolic representation of the symbol's value, and may be used in subsequent instructions to refer to the symbol's location or value.

Some assembler symbols become external symbols, and are placed in the object file's symbol table. A symbol is valid only within the module in which it is defined, unless you use the .global directive or the .def directive to declare it as an *external symbol* (see .global directive).

See Section 2.6 for more about symbols and the symbol tables in object files.

4.8.1 Identifiers

Identifiers are names used as labels, registers, symbols, and substitution symbols. An identifier is a string of alphanumeric characters, the dollar sign, and underscores (A-Z, a-z, 0-9, $, and _). The first character in an identifier cannot be a number, and identifiers cannot contain embedded blanks. The identifiers you define are case sensitive; for example, the assembler recognizes ABC, Abc, and abc as three distinct identifiers.

4.8.2 Labels

An identifier used as a label becomes an assembler symbol, which represent an address in the program. Labels within a file must be unique.

NOTE: A mnemonic cannot begin in column 1 or it will be interpreted as a label. Mnemonic opcodes and assembler directive names without the . prefix are valid label names. Remember to always use whitespace before the mnemonic, or the assembler will think the identifier is a new label definition.

Symbols derived from labels can also be used as the operands of .bss, .global, .ref, or .def directives.

```
        .global label1

label2: NOP
        ADD   AR1, label1
        SB    label2, UNC
```

4.8.3 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 4-1.
- *name*?, where *name* is any legal identifier 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. See Example 4-2.

You cannot declare these types of labels as global.

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

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

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

Example 4-1. Local Labels of the Form $n

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

```
$1:
      ADDB  AL, #-7
      B     $1, GEQ

      .newblock              ; undefine $1 to use it again.

$1    MOV   T, AL
      MPYB  ACC, T, #7
      CMP   AL, #1000
      B        $1, LT
```

The following code uses a local label illegally:

```
$1:
      ADDB  AL, #-7
      B     $1, GEQ

$1    MOV   T, AL         ; WRONG - $1 is multiply defined.
      MPYB  ACC, T, #7
      CMP   AL, #1000
      B     $1, LT
```

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.

Example 4-2. Local Labels of the Form name?

```
*****************************************************************
** First definition of local label mylab                       **
*****************************************************************
        nop
mylab?  nop
        B mylab?, UNC

*****************************************************************
** Include file has second definition of mylab                 **
*****************************************************************
        .copy  "a.inc"

*****************************************************************
** Third definition of mylab, reset upon exit from .include    **
*****************************************************************
mylab?  nop
        B mylab?, UNC

*****************************************************************
** Fourth definition of mylab in macro, macros use different   **
** namespace to avoid conflicts                                **
*****************************************************************
mymac  .macro
mylab?  nop
        B mylab?, UNC
        .endm

*****************************************************************
** Macro invocation                                            **
*****************************************************************
        mymac

*****************************************************************
** Reference to third definition of mylab. Definition is not   **
** reset by macro invocation.                                  **
*****************************************************************
        B mylab?, UNC

*****************************************************************
** Changing section, allowing fifth definition of mylab        **
*****************************************************************
        .sect  "Sect_One"
        nop
mylab?  .word 0
        nop
        nop
        B mylab?, UNC

*****************************************************************
** The .newblock directive allows sixth definition of mylab    **
*****************************************************************
        .newblock
mylab?  .word 0
        nop
        nop
        B mylab?, UNC
```

For more information about using labels in macros see Section 6.6.

4.8.4 Symbolic Constants

A symbolic constant is a symbol with a value that is an absolute constant expression (see Section 4.9). By using symbolic constants, you can assign meaningful names to constant expressions. The .set and .struct/.tag/.endstruct directives enable you to set symbolic constants (see Define Assembly-Time Constant). Once defined, symbolic constants *cannot* be redefined.

If you use the .set directive to assign a value to a symbol , the symbol becomes a symbolic constant and may be used where a constant expression is expected. For example:

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

You can also use the .set directive to assign symbolic constants for other symbols, such as register names. In this case, the symbolic constant becomes a synonym for the register:

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

The following example shows how the .set directive can be used with the .struct, .tag. and .endstruct directives. It creates the symbolic constants K, maxbuf, item, value, delta, and i_len.

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

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

array  .tag  item
array  .usect ".ebss", i_len*K  ; declare an array of K "items"
       .text
       MOV   array.delta, AR1   ; access array .delta
```

The assembler also has many predefined symbolic constants; these are discussed in Section 4.8.6.

4.8.5 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:

cl2000 --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 with assembly directives and instructions as if it had been defined with the .set directive. For example, on the command line you enter:

```
cl2000 --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 4-3 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 these 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 4-3. Using Symbolic Constants Defined on Command Line

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

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

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

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

4.8.6 Predefined Symbolic Constants

The assembler has several types of predefined symbols.

$, the dollar-sign character, represents the current value of the section program counter (SPC). $ is a relocatable symbol if you are using COFF.

In addition, the following predefined processor symbolic constants are available:

Table 4-2. C28x Processor Symbolic Constants

Symbol name	Description
__TI_EABI__	Set to 1 if EABI is enabled. Otherwise, COFF is used.
.TMS320C2000	Always set to 1
.TMS320C2800	Set to 1 for C28x
.TMS320C2800_CLA0	Set to 1 if --cla_support=cla0 or greater is used.
.TMS320C2800_CLA1	Set to 1 if --cla_support=cla1 or greater is used.
.TMS320C2800_CLA2	Set to 1 if --cla_support=cla2 is used.
.TMS320C2800_FPU32	Set to 1 if --float_support=fpu32 is used, otherwise 0.
.TMS320C2800_FPU64	Set to 1 if --float_support=fpu64 is used, otherwise 0.
.TMS320C2800_IDIV	Set to 1 if --idiv_support=idiv0 is used.
.TMS320C2800_TMU0	Set to 1 if --tmu_support=tmu0 or greater is used.
.TMS320C2800_TMU1	Set to 1 if --tmu_support=tmu1 or greater is used.
.TMS320C2800_VCU0	Set to 1 if --vcu_support=vcu0 or greater is used.
.TMS320C2800_VCU2	Set to 1 if --vcu_support=vcu2 or greater is used.
.TMS320C2800_VCRC	Set to 1 if --vcu_support=vcrc is used.

4.8.7 Registers

The names of C28x registers are predefined symbols.

In addition, control register names are predefined symbols.

Register symbols and aliases can be entered as all uppercase or all lowercase characters.

Control register symbols can be entered in all upper-case or all lower-case characters. For example, IER can also be entered as ier.

See the "Register Conventions" section of the *TMS320C28x Optimizing C/C++ Compiler User's Guide* for details about the registers and their uses.

Table 4-3. CPU and CPU Control Registers

Register	Description
ACC/AH, AL	Accumulator/accumulator high, accumulator low
DBGIER	Debug interrupt enable register
DP	Data page pointer
IER	Interrupt enable register
IFR	Interrupt flag pointer
P/PH, PL	Product register/product high, product low
PC	Program counter
RPC	Return program counter
ST0	Status register 0
ST1	Status register 1
SP	Stack pointer register
TH	Multiplicant high register; an alias of T register
XAR0/AR0H, AR0	Auxiliary register 0/auxiliary 0 high, auxiliary 0 low
XAR1/AR1H, AR1	Auxiliary register 1/auxiliary 1 high, auxiliary 1 low
XAR2/AR2H, AR2	Auxiliary register 2/auxiliary 2 high, auxiliary 2 low
XAR3/AR3H, AR3	Auxiliary register 3/auxiliary 3 high, auxiliary 3 low
XAR4/AR4H, AR4	Auxiliary register 4/auxiliary 4 high, auxiliary 4 low
XAR5/AR5H, AR5	Auxiliary register 5/auxiliary 5 high, auxiliary 5 low
XAR6/AR6H, AR6	Auxiliary register 6/auxiliary 6 high, auxiliary 6 low
XAR7/AR7H, AR7	Auxiliary register 7/auxiliary 7 high, auxiliary 7 low
XT/T, TL	Multiplicand register/Multiplicant high, multiplicant low

Table 4-4. FPU and FPU Control Registers

Register	Description
R0H	Floating point register 0
R1H	Floating point register 1
R2H	Floating point register 2
R3H	Floating point register 3
R4H	Floating point register 4
R5H	Floating point register 5
R6H	Floating point register 6
R7H	Floating point register 7
STF	Floating point status register
RB	Repeat block register

Table 4-5. VCU Registers

Register	Description
VSTATUS	VCU status and control register
VR0-VR8	VCU registers
VT0, VT1	VCU transition bit registers
VCRC	VCU CRC result register

4.8.8 Substitution Symbols

Symbols can be assigned a string value. This enables you to create aliases for 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    "AR1",  myReg         ;register AR1
    .asg    "*+XAR2 [2]",  ARG1   ;first arg
    .asg    "*+XAR2 [1]",  ARG2   ;second arg
```

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 A, B ; add2 macro definition
     MOV     AL, A
     ADD     AL, B
     .endm

*add2 invocation
      add2 LOC1, LOC2      ;add "LOC1" argument to a
                           ;second argument "LOC2".
      MOV   AL,LOC1
      ADD   AL,LOC2
```

See Chapter 6 for more information about macros.

4.9 Expressions

Nearly all values and operands in assembly language are *expressions*, which may be any of the following:

- a literal constant
- a register
- a register pair
- a memory reference
- a symbol
- a built-in function invocation
- a mathematical or logical operation on one or more expressions

This section defines several types of expressions that are referred to throughout this document. Some instruction operands accept limited types of expressions. For example, the .if directive requires its operand be an absolute constant expression with an integer value. Absolute in the context of assembly code means that the value of the expression must be known at assembly time.

A *constant expression* is any expression that does not in any way refer to a register or memory reference. An *immediate operand* will usually not accept a register or memory reference. It must be given a constant expression. Constant expressions may be any of the following:

- a literal constant
- an address constant expression
- a symbol whose value is a constant expression
- a built-in function invocation on a constant expression
- a mathematical or logical operation on one or more constant expressions

An *address constant expression* is a special case of a constant expression. Some immediate operands that require an address value can accept a symbol plus an addend; for example, some branch instructions. The symbol must have a value that is an address, and it may be an external symbol. The addend must be an absolute constant expression with an integer value. For example, a valid address constant expression is "array+4".

A constant expression may be absolute or relocatable. *Absolute* means known at assembly time. *Relocatable* means constant, but not known until link time. External symbols are relocatable, even if they refer to a symbol defined in the same module.

An *absolute constant expression* may not refer to any external symbols anywhere in the expression. In other words, an absolute constant expression may be any of the following:

- a literal constant
- an absolute address constant expression
- a symbol whose value is an absolute constant expression
- a built-in function invocation whose arguments are all absolute constant expressions
- a mathematical or logical operation on one or more absolute constant expressions

A *relocatable constant expression* refers to at least one external symbol. For ELF, such expressions may contain at most one external symbol. A relocatable constant expression may be any of the following:

- an external symbol
- a relocatable address constant expression
- a symbol whose value is a relocatable constant expression
- a built-in function invocation with any arguments that are relocatable constant expressions
- a mathematical or logical operation on one or more expressions, at least one of which is a relocatable constant expression

In some cases, the value of a relocatable address expression may be known at assembly time. For example, a relative displacement branch may branch to a label defined in the same section.

4.9.1 Mathematical and Logical Operators

The operands of a mathematical or logical operator must be well-defined expressions. That is, you must use the correct number of operands and the operation must make sense. For example, you cannot take the XOR of a floating-point value. In addition, well-defined expressions contain only symbols or assembly-time constants that have been defined before they occur in the directive's expression.

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 4-6, 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

Table 4-6 lists the operators that can be used in expressions, according to precedence group.

Table 4-6. 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.

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

4.9.2 Relational Operators and 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.

4.9.3 Well-Defined Expressions

Some assembler directives, such as .if, require well-defined absolute constant expressions as operands. Well-defined expressions contain only symbols or assembly-time constants that have been defined before they occur in the directive's expression. In addition, they must use the correct number of operands and the operation must make sense. The evaluation of a well-defined expression must be unambiguous.

This is an example of a well-defined expression:

```
1000h+X
```

where X was previously defined as an absolute symbol.

4.9.4 Legal Expressions

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

When an expression contains more than one relocatable symbol or cannot be evaluated during assembly, the assembler encodes a relocation expression in the object file that is later evaluated by the linker. If the final value of the expression is larger in bits than the space reserved for it, you receive an error message from the linker. See Section 2.7 for more information on relocation expressions.

When using the register relative addressing mode, the expression in brackets or parenthesis must be a well-defined expression, as described in Section 4.9.3. For example:

```
*+XA4[7]
```

4.10 Built-in Functions and Operators

The assembler supports built-in mathematical functions and built-in addressing operators.

The built-in substitution symbol functions are discussed in Section 6.3.2.

4.10.1 Built-In Math and Trigonometric Functions

The assembler supports many built-in mathematical functions. The built-in functions always return a value and they can be used in conditional assembly or any place where a constant can be used.

In Table 4-7 x, y and z are type float, n is an int. The functions $cvi, $int and $sgn return an integer and all other functions return a float. Angles for trigonometric functions are expressed in radians.

Table 4-7. Built-In Mathematical Functions

Function	Description
$acos(*x*)	Returns $\cos^{-1}(x)$ in range [0, π], -1<=*x*<=1
$asin(*x*)	Returns $\sin^{-1}(x)$ in range [-π/2, π/2], -1<=*x*<=1
$atan *x*)	Returns $\tan^{-1}(x)$ in range [-π/2, π/2]
$atan2(*x*, *y*)	Returns $\tan^{-1}(y/x)$ in range [-π, π]
$ceil(*x*)	Returns the smallest integer not less than *x*, as a float
$cos(*x*)	Returns the cosine of *x*
$cosh(*x*)	Returns the hyperbolic cosine of *x*
$cvf(*n*)	Converts an integer to a float
$cvi(*x*)	Converts a float to an integer. Returns an integer.
$exp(*x*)	Returns the exponential function e^x
$fabs(*x*)	Returns the absolute value |*x*|
$floor(*x*)	Returns the largest integer not greater than *x*, as a float
$fmod(*x*, *y*)	Returns the floating-point remainder of *x*/*y*, with the same sign as *x*
$int(*x*)	Returns 1 if *x* has an integer value; else returns 0. Returns an integer.
$ldexp(*x*, *n*)	Multiplies *x* by an integer power of 2. That is, $x \times 2^n$
$log(*x*)	Returns the natural logarithm ln(*x*), where *x*>0
$log10(*x*)	Returns the base-10 logarithm $\log_{10}(x)$, where *x*>0
$max(*x*, *y*, ...*z*)	Returns the greatest value from the argument list
$min(*x*, *y*, ...*z*)	Returns the smallest value from the argument list
$pow(*x*, *y*)	Returns x^y
$round(*x*)	Returns *x* rounded to the nearest integer
$sgn(*x*)	Returns the sign of *x*. Returns 1 if *x* is positive, 0 if *x* is zero, and -1 if *x* is negative. Returns an integer.
$sin(*x*)	Returns the sine of *x*
$sinh(*x*)	Returns the hyperbolic sine of *x*
$sqrt(*x*)	Returns the square root of x, x≥0
$strtod(*str*)	Converts a character string to a double precision floating-point value. The string contains a properly-formatted C99-style floating-point literal.
$tan(*x*)	Returns the tangent of *x*
$tanh(*x*)	Returns the hyperbolic tangent of *x*
$trunc(*x*)	Returns *x* truncated toward 0

4.11 TMS320C28x Assembler Extensions

The C28x assembler operates with a variety of extensions. These extensions are controlled by options as follows:

-v28: The -v28 is the default, and no other silicon version is supported for the -v option. Therefore you do not need to specify -v28 explicitly.

--cla_support: Accept CLA instructions. To support special floating point instructions that run on the Control Law Accelerator (CLA), the assembler operates with CLA extensions. Section 4.11.3 describes the CLA support. CLA Type 0, Type 1, or Type 2 can be specified. This extension is controlled by the following option: --cla_support=[cla0|cla1|cla2].

--float_support: Accept FPU32 or FPU64 instructions. Support is available for special floating point instructions when a 32-bit or 64-bit floating point unit (FPU) is available. Section 4.11.2 describes the FPU support. This extension is controlled by the following option: --float_support=[fpu32|fpu64].

--idiv_support: Substitute fast integer division instructions. These instructions support division and modulo operations. This extension is controlled by the following option: --idiv_support=idiv0.

--tmu_support: Substitute TMU instructions. To support the Trigonometric Math Unit (TMU), TMU instructions are used for floating point division and trigonometric functions. The options are tmu0 (default) and tmu1. The tmu1 setting provides support for all tmu0 functionality plus the LOG2F32 and IEXP2F32 instructions. This extension is controlled by the following option: --tmu_support=[tmu0|tmu1].

--vcu_support: Accept VCU instructions. To support the Viterbi, Complex Math and CRC Unit (VCU) instructions, the assembler operates with VCU support. The VCU Type 0, Type 2, or Cyclic Redundancy Check (CRC) option can be specified. This extension is controlled by the following option: --vcu_support=[vcu0|vcu2|vcrc].

Refer to the *TMS320C28x DSP CPU and Instruction Set Reference Guide* for more details on the different object and addressing extensions supported by the C28x processor.

4.11.1 C28x Support

The default support includes all the C28x instructions and generates C28x object code.

An error occurs if old C27x syntax is used. For example, the following instructions are illegal:

```
MOV    AL, *AR0++    ; *AR0++ is illegal addressing for C28x.
```

4.11.2 C28x FPU32 and FPU64 Extensions

FPU instructions may be used when the 32-bit or 64-bit floating-point co-processor hardware is available on the C28x.

FPU32 instructions are enabled by specifying the --float_support=fpu32 option. FPU32 instructions are supported for both the COFF ABI and EABI.

FPU64 instructions are enabled by specifying the --float_support=fpu64 option. The FPU64 instruction set includes all of the FPU32 instructions plus some additional 64-bit instructions. FPU64 is supported only for EABI.

When FPU support is enabled, the differences are as follows:

- Some special floating point instructions are supported. These are documented in the *TMS320C28x Floating Point Unit and Instruction Set Reference Guide*.
- The assembler checks for pipeline conflicts. This is because the FPU instructions are not pipeline protected. Regular C28x instructions are pipeline protected, which means that a new instruction cannot read/write its operands until all preceding C28x instructions have finished writing those operands. This is not the case with FPU instructions, which can access operands while another instruction is writing them, causing race conditions. Thus the assembler has to check for pipeline conflicts and issue warnings/errors as appropriate. The pipeline conflict detection feature is described in Section 4.16.

4.11.3 C28x CLA Extensions

A set of instructions to support the Control Law Accelerator (CLA) co-processor is available for the C28x. These instructions are enabled by invoking the compiler with the --cla_support=[cla0|cla1|cla2] option. Using cla0 indicates a CLA Type 0 device, cla1 indicates a Type 1 device, and so on. The --cla_support option can be specified along with other C28x options, such as those for specifying FPU support. Specifying both FPU and CLA options means that support is available for both types of accelerators. The CLA support is similar to the base C28x support (with/without FPU support). The differences are:

- The CLA is similar to a cut-down version of the FPU32 that is optimized to perform math tasks only. Some special floating point instructions are supported. These are documented in *TMS320x28xx, 28xxx DSP Peripherals Reference Guide*.
- The CLA pipeline is unprotected, but at this time, the tools do not detect pipeline conflicts for the CLA. You need to write CLA instructions in such a way that there are no pipeline conflicts.
- Assembly files containing CLA instructions can also contain C28x and FPU instructions. However, the CLA instructions should always be in a separate, named section. This section cannot contain any non-CLA instructions. Mixing CLA and non-CLA instructions in the same section is illegal and results in an assembler/linker error.
- When a linker command file is written, care must be taken to put all data referenced by CLA instructions within addresses 0-64K. This is because the CLA data read bus only has a 64K address range.
- A linker output section containing a CLA input section cannot contain any non-CLA input sections.
- The CLA support does not need any special library support. Any of the C28x libraries suffices.
- The name of the section containing CLA instructions should be unique both within the file and across all files that are compiled and linked into the same output file.

The CLA compiler places all CLA function data, arguments, and temporary storage in function frames in the .scratchpad section. Function frame scratchpad sections are named in the form ".scratchpad:*functionSectionName*". (Each function has its own subsection and therefore a unique section name.) For example: .scratchpad:Cla1Prog:_Cla1Task2 would be the compiler-generated scratchpad section name for a function called Cla1Task2().

The CLA compiler's naming convention for the function scratchpad symbol is of the form __cla_*functionSymbol*_sp, but this is not required in assembly code.

CLA2 background tasks are placed in the scratchpad with function frame sections of the form ".scratchpad:background:*functionSectionName*". The background task frame cannot be overlaid with any other function frames, since the background task is likely to be returned to after yielding to interrupts.

The following example shows compiled code with a .sect directive for a CLA function and a .usect directive to identify the function scratchpad frame. This .usect directive identifies the function frame as part of the .scratchpad section and allows the compiler to use overlays when possible. Overlaid function frames use the same physical memory, thereby reducing memory utilization. It is recommended that assembly code follow the ".scratchpad:" naming convention to reduce memory requirements.

```
        .sect "Cla1Prog:_Cla1Task2"
        .align 2
__cla_Cla1Task2_sp .usect ".scratchpad:Cla1Prog:_Cla1Task2",14,0,1
        .global _Cla1Task2

;*************************************************************
;* FNAME: _Cla1Task2                    FR SIZE:  14           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                           14 Auto, 12 SOE     *
;*************************************************************

_Cla1Task2:
        [ ... ]
```

If the CLA function in the example above were a CLA2 background task, the .usect directive would instead identify the scratchpad frame as ".scratchpad:background:Cla1Prog:_Cla1Task2".

See the "CLA Compiler" chapter in the *TMS320C28x Optimizing C/C++ Compiler User's Guide* for more details.

4.12 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 4.3).

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 4-2 shows these in an actual listing file.

Field 1: Source Statement Number

Line number

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

Include file letter

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

Nesting level number

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

Field 2: Section Program Counter

This field contains the SPC value, which is hexadecimal. All sections (.text, .data, .bss, .ebss, 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:

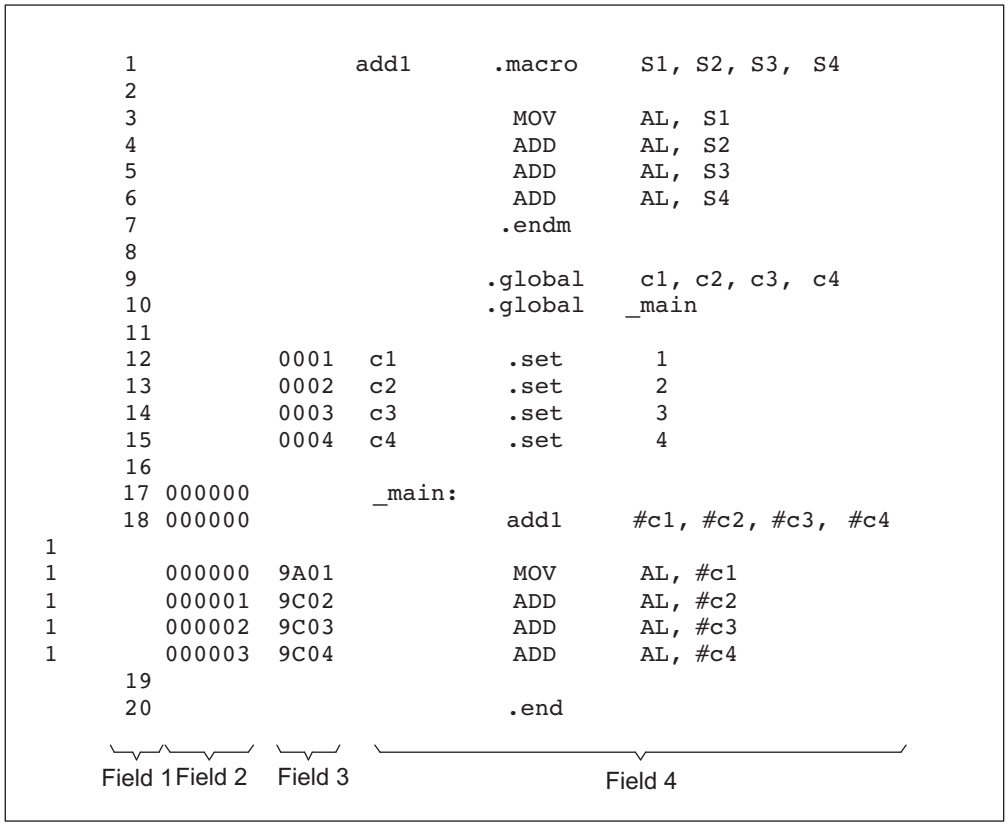
!	undefined external reference
'	.text relocatable
+	.sect relocatable
"	.data relocatable
-	.bss, .usect relocatable
%	relocation expression

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 4-2 shows an assembler listing with each of the four fields identified.

Figure 4-2. Example Assembler Listing



4.13 Debugging Assembly Source

By default, when you compile 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 default has the same behavior as using the --symdebug:dwarf option. You can disable the generation of debugging information by using the --symdebug:none option.

The .asmfunc and .endasmfunc (see .asmfunc directive) 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 .ref directive). Example 4-4 shows the cvar.c C program that defines a variable, svar, as the structure type X. The svar variable is then referenced in the addfive.asm assembly program in Example 4-5 and 5 is added to svar's second data member.

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

```
cl2000 -symdebug:dwarf cvars.c addfive.asm --run_linker --library=lnk.cmd
     --library=rts2800_ml.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 4-4. Viewing Assembly Variables as C Types C Program

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

Example 4-5. Assembly Program for Example 4-4

```
;-------------------------------------------------------------------------
; Tell assembler we reference variable _svar, which is defined in cvars.c file.
;-------------------------------------------------------------------------
          .ref _svar
;-------------------------------------------------------------------------
; addfive() - Add five to the second data member of _svar
;-------------------------------------------------------------------------
          .text
          .global addfive
addfive:  .asmfunc
           MOVZ    DP,#_svar+1    ; load the DP with svar's memory page
           ADD     @_svar+1,#5    ; add 5 to svar.m2
           LRETR                  ; return from function
          .endasmfunc
```

4.14 Cross-Reference Listings

A cross-reference listing shows symbols and their definitions. To obtain a cross-reference listing, invoke the assembler with the --asm_listing_cross_reference option (see Section 4.3) 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 4-6 shows the four fields contained in the cross-reference listing.

Example 4-6. An Assembler Cross-Reference Listing

```
LABEL                              VALUE       DEFN    REF
.TMS320C2800                    00000001          0
_func                          00000000'         18
var1                           00000000-          4     17
var2                           00000004-          5     18
```

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 4-8 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 4-8. 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

4.15 Smart Encoding

To improve efficiency, the assembler reduces instruction size whenever possible. For example, a branch instruction of two words can be changed to a short branch one-word instruction if the offset is 8 bits. Table 4-9 lists the instruction to be changed and the change that occurs.

Table 4-9. Smart Encoding for Efficiency

This instruction...	Is encoded as...
MOV AX, *#8Bit*	MOVB AX, *#8Bit*
ADD AX, *#8BitSigned*	ADDB AX, *#8BitSigned*
CMP AX, *#8Bit*	CMPB AX, *#8Bit*
ADD ACC, *#8Bit*	ADDB ACC, *#8Bit*
SUB ACC, *#8Bit*	SUBB ACC, *#8Bit*
AND AX, *#8BitMask*	ANDB AX, *#8BitMask*
OR AX, *#8BitMask*	ORB AX, *#8BitMask*
XOR AX, *#8BitMask*	XORB AX, *#8BitMask*
B *8BitOffset*, *cond*	SB *8BitOffset*, *cond*
LB *8BitOffset*, *cond*	SB *8BitOffset*, *cond*
MOVH *loc*, ACC << 0	MOV *loc*, AH
MOV *loc*, ACC << 0	MOV *loc*, AL
MOVL XAR*n*, *#8Bit*	MOVB XAR*n*, *#8Bit*

The assembler also intuitively changes instruction formats during smart encoding. For example, to push the accumulator value to the stack, you use MOV *SP++, ACC. Since it would be intuitive to use PUSH ACC for this operation, the assembler accepts PUSH ACC and through smart encoding, changes it to MOV *SP++, ACC. Table 4-10 shows a list of instructions recognized during intuitive smart encoding and what the instruction is changed to.

Table 4-10. Smart Encoding Intuitively

This instruction...	Is encoded as...
MOV P, #0	MPY P, T, #0
SUB *loc*, *#16BitSigned*	ADD *loc*, *#-16BitSigned*
ADDB SP, *#-7Bit*	SUBB SP, *#7Bit*
ADDB *aux*, *#-7Bit*	SUBB *aux*, *#7Bit*
SUBB AX, *#8BitSigned*	ADDB AX, *#-8BitSigned*
PUSH IER	MOV *SP++, IER
POP IER	MOV IER, *--SP
PUSH ACC	MOV *SP++, ACC
POP ACC	MOV ACC, *--SP
PUSH XAR*n*	MOV *SP++, XAR*n*
POP XAR*n*	MOV XAR*n*, *--SP
PUSH *#16Bit*	MOV *SP++, *#16Bit*
MPY ACC, T, *#8Bit*	MPYB ACC, T, *#8Bit*

In some cases, you might want a 2-word instruction even when there is an equivalent 1-word instruction available. In such cases, smart encoding for efficiency could be a problem. Therefore, the equivalent instructions in Table 4-11 are provided; these instructions will not be optimized.

Table 4-11. Instructions That Avoid Smart Encoding

This instruction...	Is encoded as...
MOVW AX, *#8Bit*	MOV AX, *#8Bit*
ADDW AX, *#8Bit*	ADD AX, *#8Bit*
CMPW AX, *#8Bit*	CMP AX, *#8Bit*
ADDW ACC, *#8Bit*	ADD ACC, *#8Bit*
SUBW ACC, *#8Bit*	SUB ACC, *#8Bit*
JMP *8BitOffset*, *cond*	B *8BitOffset*, *cond*

4.16 Pipeline Conflict Detection

Pipeline Conflict Detection (PCD) is a feature implemented on the TMS320C28x 5.0 Compiler, for targets with hardware floating point unit (FPU) support only. This is because the FPU instructions are not pipeline protected whereas the C28x instructions are. Beginning with version 6.0, similar protections are provided for targets with support for the Viterbi, Complex Math and CRC Unit (VCU).

4.16.1 Protected and Unprotected Pipeline Instructions

The C28x target with FPU/VCU support has a mix of protected and unprotected pipeline instructions. This necessitates some checks in the compiler and assembler that are not necessary for a C28x target without such support.

By design, a (non-FPU) C28x instruction does not read/write an operand until all previous instructions have finished writing that operand. The hardware stalls until this condition is true. As hardware stalls are employed to preserve operand integrity, the compiler and assembler need not keep track of register reads and writes by instructions in the pipeline. Thus, the C28x instructions are pipeline protected, meaning that an instruction will not attempt to read/write a register while that register is still being written by another instruction.

The situation is different when FPU support is enabled. While the non-FPU instructions are pipeline protected, the FPU instructions aren't. This implies that an FPU instruction could attempt to read/write a register while it is still being written by a previous instruction. This can cause undefined behavior, and the compiler and assembler need to protect against such conflicting register accesses. The same is true for VCU instructions.

4.16.2 Pipeline Conflict Prevention and Detection

The compiler, when generating assembly code from C/C++ programs, ensures that the generated code does not have any pipeline conflicts. It does this by either scheduling non-conflicting instructions between two potentially conflicting instructions, or inserting NOP instructions wherever necessary. For details on the compiler, please see the .

While conflict prevention by the compiler is sufficient for C/C++ test cases, this does not cover manually-written assembly language code. Assembly code can contain instructions that have pipeline conflicts. The assembler needs to detect such conflicts and issue warnings or errors, depending on the severity of the situation. This is what the Pipeline Conflict Detection (PCD) feature in the assembler, is designed to do.

4.16.3 Pipeline Conflicts Detected

The assembler detects certain pipeline conflicts, and based on their severity, issues either an error message or a warning. The types of pipeline conflicts detected are listed below, along with the assembler actions in the event of each conflict.

- **Pipeline Conflict:**

  An instruction reads a register when it is being written by another instruction.

  Assembler Response:

  The assembler generates an error message and aborts.
- **Pipeline Conflict:**

  Two instructions write the same register in the same cycle.

  Assembler Response:

  The assembler generates an error message and aborts.
- **Pipeline Conflict:**

  Instructions FRACF32, I16TOF32, UI16TOF32, F32TOI32, and/or F32TOUI32 are present in the delay slot of a specific type of MOV32 instruction that moves a value from a CPU register or memory location to an FPU register.

  Assembler Response:

  The assembler gives an error message and aborts, as the hardware is not able to correctly execute this sequence.
- **Pipeline Conflict:**

  Parallel operations have the same destination register.

  Assembler Response:

  The assembler gives a warning.
- **Pipeline Conflict:**

  A read/write happens in the delay slot of a write of the same register.

  Assembler Response:

  The assembler gives a warning.
- **Pipeline Conflict:**

  A SAVE operation happens in the delay slot of a pipeline operation.

  Assembler Response:

  The assembler gives a warning.
- **Pipeline Conflict:**

  A RESTORE operation happens in the delay slot of a pipeline operation.

  Assembler Response:

  The assembler gives a warning.
- **Pipeline Conflict:**

  A SETFLG instruction tries to modify the LUF or LVF flag while certain instructions that modify LUF/LVF (such as ADDF32, SUBF32, EINVF32, EISQRTF32 etc) have pending writes.

  Assembler Response:

  The assembler does not check for which instructions have pending writes; on encountering a SETFLG when any write is pending, the assembler issues a detailed warning, asking you to ensure that the SETFLG is not in the delay slot of the specified instructions.

For the actual timing of each FPU instruction, and pipeline modeling, please refer to the *TMS320C28x Floating Point Unit and Instruction Set Reference Guide*. Timing information for VCU instructions can be found in the *TMS320x28xx, 28xxx DSP Peripherals Reference Guide*.

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 5.1 through Section 5.11) describes the directives according to function, and the second part (Section 5.12) is an alphabetical reference.

Topic **Page**

5.1 Directives Summary

Table 5-1 through Table 5-16 summarize the assembler directives.

Besides the assembler directives documented here, the TMS320C28x software tools support the following directives:

- Macro directives are discussed in Chapter 6; 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: Most source statements that contain 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 here. See the detailed description of each directive for using labels with directives.

Table 5-1. Directives that Control Section Use

Mnemonic and Syntax	Description	See
.bss *symbol, size in words* [**,** *blocking flag* [**,** *alignment*]]	Reserves *size* words in the .bss (uninitialized data) section	.bss 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 words* [**,** *blocking flag*[**,** *alignment flag*]]	Reserves *size* words in a named (uninitialized) section	.usect topic

Table 5-2. Directives that Gather Sections into Common Groups

Mnemonic and Syntax	Description	See
.endgroup	Ends the group declaration. (EABI only)	.endgroup topic
.gmember *section name*	Designates *section name* as a member of the group. (EABI only)	.gmember topic
.group *group section name group type* **:**	Begins a group declaration. (EABI only)	.group topic

Table 5-3. Directives that Affect Unused Section Elimination

Mnemonic and Syntax	Description	See
.clink "*section name***"**	Enables conditional linking for the current or specified section. (COFF only)	.clink topic
.retain "*section name***"**	Instructs the linker to include the current or specified section in the linked output file, regardless of whether the section is referenced or not. (EABI only)	.retain topic
.retainrefs "*section name***"**	Instructs the linker to include any data object that references the current or specified section. (EABI only)	.retain topic

Table 5-4. Directives that Initialize Values (Data and Memory)

Mnemonic and Syntax	Description	See
.bits $value_1$**[,** ... **,** $value_n$]	Initializes one or more successive bits in the current section	.bits topic
.byte $value_1$**[,** ... **,** $value_n$]	Initializes one or more successive words in the current section	.byte topic
.char $value_1$**[,** ... **,** $value_n$]	Initializes one or more successive words in the current section	.char topic

Table 5-4. Directives that Initialize Values (Data and Memory) (continued)

Mnemonic and Syntax	Description	See
.cstring {*expr*$_1$|"*string*$_1$"}[,... , {*expr*$_n$|"*string*$_n$"}]	Initializes one or more text strings	.string 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
.int *value*$_1$[**,** ... **,** *value*$_n$]	Initializes one or more 16-bit integers	.int topic
.long *value*$_1$[**,** ... **,** *value*$_n$]	Initializes one or more 32-bit integers	.long topic
.pstring {*expr*$_1$|" *string*$_1$"}[,... , {*expr*$_n$|"*string*$_n$ "}]	Places 8-bit characters from a character string into the current section.	.pstring 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
.uchar *value*$_1$[**,** ... **,** *value*$_n$]	Initializes one or more successive unsigned bytes in the current section	.uchar topic
.uint *value*$_1$[**,** ... **,** *value*$_n$]	Initializes one or more unsigned 32-bit integers	.uint topic
.ulong *value*$_1$[**,** ... **,** *value*$_n$]	Initializes one or more unsigned 32-bit integers	.long 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*$_n$]	Places the 32-bit floating-point representation of one or more floating-point constants into the current section	.xfloat topic
.xldouble *value*$_1$[**,** ... **,** *value*$_n$]	Places the 64-bit floating-point representation of one or more floating-point double constants into the current section	.xfloat topic
.xlong *value*$_1$[**,** ... **,** *value*$_n$]	Places one or more 32-bit values into consecutive words in the current section	.xlong topic

Table 5-5. Directives that Perform Alignment and Reserve Space

Mnemonic and Syntax	Description	See
.align [*size in words*]	Aligns the SPC on a boundary specified by *size in words*, which must be a power of 2; defaults to 64-byte or page boundary	.align topic
.bes *size*	Reserves *size* bits in the current section; a label points to the end of the reserved space	.bes topic
.space *size*	Reserves *size* words in the current section; a label points to the beginning of the reserved space	.space topic

Table 5-6. 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

Table 5-6. Directives that Format the Output Listing (continued)

Mnemonic and Syntax	Description	See
.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 5-7. 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 5-8. Directives that Affect Symbol Linkage and Visibility

Mnemonic and Syntax	Description	See
.common *symbol, size in bytes* [, *alignment*] **.common** *symbol, structure tag* [, *alignment*]	Defines a common symbol for a variable. (EABI only)	.common topic
.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
.weak *symbol name*	Identifies a symbol used in the current module that is defined in another module. (EABI only)	.weak topic

Table 5-9. Directives that Enable Conditional Assembly

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

Table 5-10. 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 5.9
.endenum	Sets up C-like enumerated types in assembly code	Section 5.9
.endstruct	Ends a structure definition	.cstruct topic, .struct topic

Table 5-10. Directives that Define Union or Structure Types (continued)

Mnemonic and Syntax	Description	See
.endunion	Ends a union definition	.cunion topic, .union topic
.enum	Sets up C-like enumerated types in assembly code	Section 5.9
.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 5-11. Directives that Define Symbols at Assembly Time

Mnemonic and Syntax	Description	See
.asg ["]*character string*["], *substitution symbol*	Assigns a character string to *substitution symbol*. Substitution symbols created with .asg can be redefined.	.asg topic
.define ["]*character string*["], *substitution symbol*	Assigns a character string to *substitution symbol*. Substitution symbols created with .define cannot be redefined.	.asg topic
.elfsym *name*, SYM_SIZE(*size*)	Provides ELF symbol information. (EABI only)	.elfsym topic
.eval *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 5-12. Directives that Create or Affect Macros

Mnemonic and Syntax	Description	See
macname **.macro** [*parameter*$_1$][,... , *parameter*$_n$]	Begin definition of macro named *macname*	.macro topic
.endm	End macro definition	.endm topic
.mexit	Go to .endm	Section 6.2
.mlib *filename*	Identify library containing macro definitions	.mlib topic
.var	Adds a local substitution symbol to a macro's parameter list	.var topic

Table 5-13. 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
.wmsg *string*	Sends user-defined warning messages to the output device	.wmsg topic

Table 5-14. 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 5-15. Directives that Are Used by the Absolute Lister

Mnemonic and Syntax	Description	See
.setsect	Produced by absolute lister; sets a section	Chapter 9
.setsym	Produced by the absolute lister; sets a symbol	Chapter 9

Table 5-16. Directives that Perform Miscellaneous Functions

Mnemonic and Syntax	Description	See
.cdecls [*options* **,]"***filename***"[, "***filename2***"[,** ...]	Share C headers between C and assembly code	.cdecls topic
.end	Ends program	.end topic
.sblock	Designates section for blocking	.sblock topic

In addition to the assembly directives that you can use in your code, the C/C++ 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
- The **.compiler_opts** directive indicates that the assembly code was produced by the compiler, and which build model options were used for this file.

5.2 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. (COFF only)
- The **.data** directive identifies portions of code in the .data section. The .data section usually contains initialized data.
- The **.retain** directive can be used to indicate that the current or specified section must be included in the linked output. Thus even if no other sections included in the link reference the current or specified section, it is still included in the link. (EABI only)
- The **.retainrefs** directive can be used to force sections that refer to the specified section. This is useful in the case of interrupt vectors. (EABI only)
- 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 5-1 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 5-1 perform the following tasks:

.text	initializes words with the values 1, 2, 3, 4, 5, 6, 7, and 8.
.data	initializes words with the values 9, 10, 11, 12, 13, 14, 15, and 16.
var_defs	initializes words with the values 17 and 18.
.usect	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 5-1. Sections Directives

```
 1              ************************************************
 2              *     Start assembling into the .text section     *
 3              ************************************************
 4 000000               .text
 5 000000 0001          .word   1, 2
   000001 0002
 6 000002 0003          .word   3, 4
   000003 0004
 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       sym     .usect  ".ebss", 19 ; Reserve space in .ebss
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 0005          .word   5, 6
   000005 0006
35 000000       usym    .usect  "xy", 20   ; Reserve space in xy
36 000006 0007          .word   7, 8       ; Still in .text
37 000007 0008
```

5.3 Directives that Initialize Values

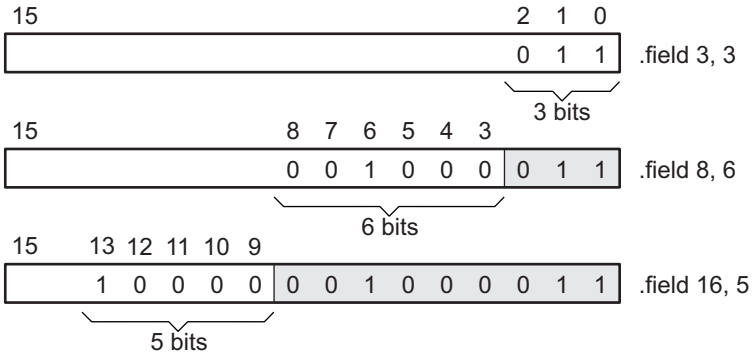
Several directives assemble values for the current section. For example:

- The **.byte** and **.char** directives place one or more 8-bit values into consecutive 16-bit words of the current section. These directives are similar to .word, .int, and .long, except that the width of each value is restricted to 8 bits.
- The **.field** and **.bits** directives place 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. If a field will not fit in the space remaining in the current word, .field will insert zeros to fill the current word and then place the field in the next word. The .bits directive is similar but does not force alignment to a field boundary. See the .field topic and .bits topic.

  Figure 5-1 shows how fields are packed into a word. Using the following assembled code, notice that the SPC does not change (the fields are packed into the same word):

```
1 000000 0003          .field  3, 3
2 000000 0008          .field  8, 6
3 000000 0010          .field  16, 5
```

Figure 5-1. 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 **.int** and **.word** 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 **.long** and **.xlong** directives place one or more 32-bit values into consecutive 32-bit fields (words) in the current section. The .long directive automatically aligns to a word boundary.
- The **.xldouble** directive calculates the double-precision (64-bit) IEEE floating-point representation of a double floating-point value and stores it into consecutive 32-bit fields (words) in the current section that is aligned to a word boundary. Note that the **.double** directive is not a synonym and is not recommended.
- 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 16-bit word of the current section. The .cstring directive adds a NUL character needed by C; the .string directive does not add a NUL character. With the .pstring directive, the data is packed so that each word contains two 8-bit bytes.
- The **.ubyte**, **.uchar**, **.uint**, **.ulong**, and **.uword** directives are provided as unsigned versions of their respective signed directives. These directives are used primarily by the C/C++ compiler to support unsigned types in C/C++.

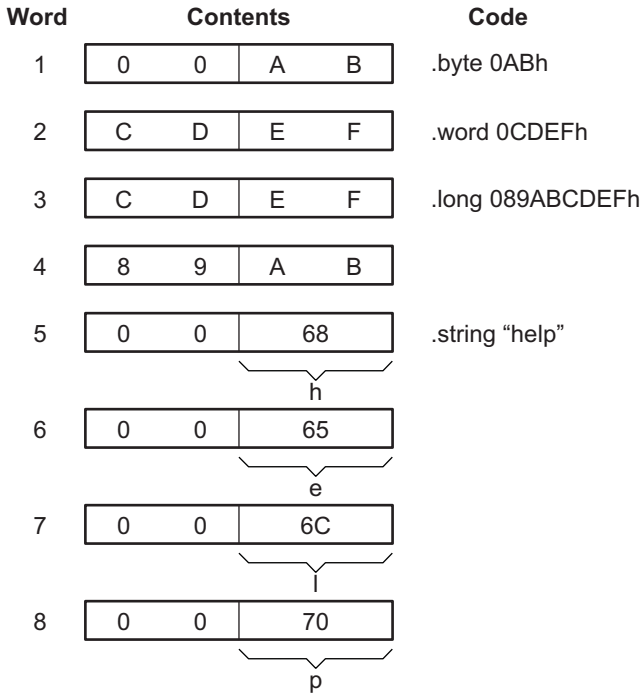
Directives that Initialize Constants When Used in a .struct/.endstruct Sequence

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

Figure 5-2 compares the .byte, .word, .long, and .string directives using the following assembled code:

```
1 000000 00AB          .byte   0ABh
2 000001 CDEF          .word   0CDEFh
3 000002 CDEF          .long   089ABCDEFh
  000003 89AB
4 000004 0068          .string "help"
  000005 0065
  000006 006C
  000007 0070
```

Figure 5-2. Initialization Directives



5.4 Directives that Perform Alignment and Reserve Space

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

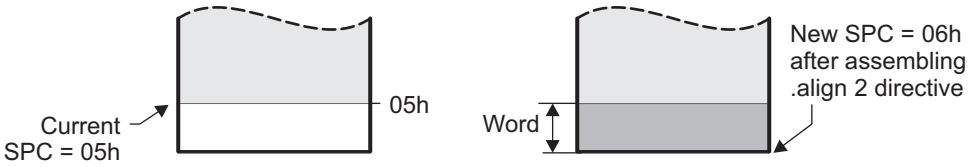
- The **.align** directive aligns the SPC at the next word boundary. This directive is useful with the .field directive when you do not want to pack two adjacent fields in the same word.

  Figure 5-3 demonstrates the .align directive. Using the following assembled code:

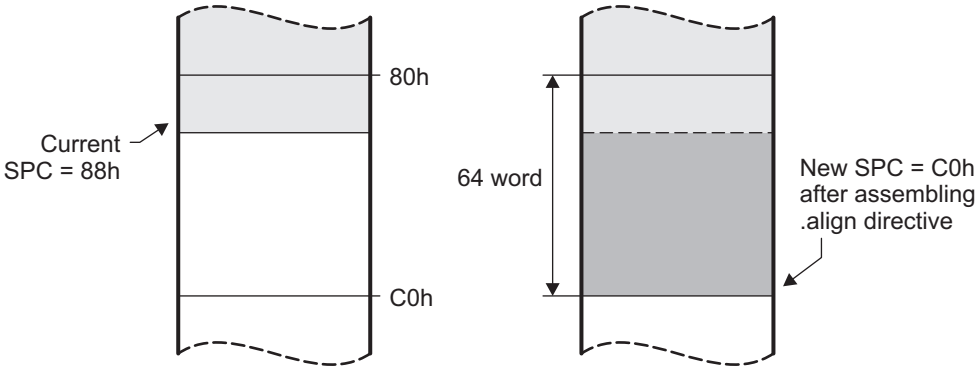
```
1 000000 0002          .field  2,3
2 000000 005A          .field  11,8
3                      .align  2
4 000002 0065          .string "errorcnt"
  000003 0072
  000004 0072
  000005 006F
  000006 0072
  000007 0063
  000008 006E
  000009 0074
5                      .align
6 000040 0004          .byte   4
```

Figure 5-3. The .align Directive

(a) Result of .align 2



(b) Result of .align without an argument



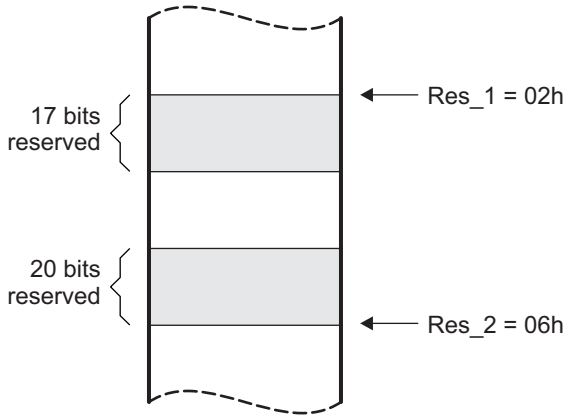
- The **.bes** and **.space** directives reserve a specified number of bits in the current section. The assembler fills these reserved bits with 0s.
  - When you use a label with .space, it points to the *first* word that contains reserved bits.
  - When you use a label with .bes, it points to the *last*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

  Figure 5-4 shows how the .space and .bes directives work for the following assembled code:

```
1
2
3 000000 0100          .word   100h, 200h
  000001 0200
4 000002       Res_1   .space  17
5 000004 000F          .word   15
6 000006       Res_2   .bes    20
7 000007 00BA          .byte   0BAh
```

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

Figure 5-4. The .space and .bes Directives



5.5 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).
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, L, M, T, and W directives (turns off the limits of B, L, 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 --asm_listing_cross_reference option (see Section 4.3).

- 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.

5.6 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 2.6.1). 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.
- The **.weak** directive identifies a symbol that is used in the current module but is defined in another module. It is equivalent to the .ref directive, except that the reference has weak linkage. (EABI only)

5.7 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 *condition*	marks the beginning of a conditional block and assembles code if the .if *condition* is true.
[**.elseif** *condition*]	marks a block of code to be assembled if the .if *condition* is false and the .elseif condition is true.
.else	marks a block of code to be assembled if the .if *condition* 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 [*count*]	marks the beginning of a repeatable block of code. The optional expression evaluates to the loop count.
.break [*end condition*]	tells the assembler to assemble repeatedly when the .break *end condition* 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 4.9.2.

5.8 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.

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 13. For .struct and .union, element offset calculation is left up to the assembler, so the layout may be different than .cstruct and .cunion.

5.9 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 13.

See Section 13.2.10 for an example of using .enum.

5.10 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 created with .asg 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 **.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 **.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 **.set** directive sets 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 directive produces no object code.
- 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.

5.11 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 **.group**, **.gmember**, and **.endgroup** directives define an ELF group section to be shared by several sections. (EABI only)
- 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 4.8.3 for information on local labels.
- The **.sblock** directive designates sections for blocking.

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 6.7.

5.12 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 words*]

Description

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size in words* parameter. The *size* can be any power of 2, although only certain values are useful for alignment. An operand of 64 aligns the SPC on the next page boundary, and this is the default if no *size in words* is given. The assembler assembles words containing null values (0) up to the next *size in words* boundary:

1	aligns SPC to word boundary
2	aligns SPC to long word/even boundary
64	aligns SPC to page boundary

Using the .align directive has two effects:

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

Example

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

```
 1 000000 0004          .byte     4
 2                      .align    2
 3 000002 0045          .string   "Errorcnt"
   000003 0072
   000004 0072
   000005 006F
   000006 0072
   000007 0063
   000008 006E
   000009 0074
 4                      .align
 5 000040 0003          .field    3,3
 6 000040 002B          .field    5,4
 7                      .align    2
 8 000042 0003          .field    3,3
 9                      .align    8
10 000048 0005          .field    5,4
11                      .align
12 000080 0004          .byte     4
```

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

Syntax

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

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

.eval *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 128 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

The **.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 13 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 *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 *expression* is a well-defined alphanumeric expression in which all symbols have been previously defined in the current source module, so that the result is an absolute expression.
- The *substitution symbol* must be a valid symbol name. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

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
      2                       .asg    XAR6, FP
      3 00000000 0964           ADD     ACC, #100
      4 00000001 7786           NOP     *FP++
#                               NOP     *XAR6++
      5 00000002 7786           NOP     *XAR6++
      6
      7                       .asg    0, x
      8                       .loop   5
      9                       .eval   x+1, x
     10                       .word   x
     11                       .endloop
1                             .eval   x+1, x
#                             .eval   0+1, x
1        00000003 0001          .word   x
#                             .word   1
1                             .eval   x+1, x
#                             .eval   1+1, x
1        00000004 0002          .word   x
#                             .word   2
1                             .eval   x+1, x
#                             .eval   2+1, x
1        00000005 0003          .word   x
#                             .word   3
1                             .eval   x+1, x
#                             .eval   3+1, x
1        00000006 0004          .word   x
#                             .word   4
1                             .eval   x+1, x
#                             .eval   4+1, x
1        00000007 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 *symbol* is a label that must appear in the label field.

The .asmfunc directive has an optional parameter, stack_usage, which indicates that the function may use up 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 userfunc section.

```
 1 00000000                    .sect       ".text"
 2                             .global     userfunc
 3                             .global     _printf
 4
 5                userfunc:    .asmfunc
 6 00000000 FE02               ADDB      SP,#2
   00000002 0000
 8 00000003 7640!              LCR       #_printf
   00000004 0000
 9 00000005 9A00               MOVB      AL,#0
10 00000006 FE82               SUBB      SP,#2
11 00000007 0006               LRETR
12                             .endasmfunc
13
14 00000000                    .sect       ".econst"
15 00000000 0048  SL1:         .string     "Hello World!",10,0
   00000001 0065
   00000002 006C
   00000003 006C
   00000004 006F
   00000005 0020
   00000006 0057
   00000007 006F
   00000008 0072
   00000009 006C
   0000000a 0064
   0000000b 0021
   0000000c 000A
   0000000d 0000
```

.bits *Initialize Bits*

Syntax

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

Description

The **.bits** directive places a value into consecutive bits of the current section.

The .bits directive is similar to the .field directive (see .field topic). However, the .bits directive does not force the value to be aligned to a field boundary. If the .bits directive is followed by a different space-creating directive, the SPC is aligned to an appropriate value for the directive that follows.

This directive has two operands:

- The *value* is a required parameter; it is an expression that is evaluated and placed in the current section at the current location. 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 value. The default 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, .bits 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
                .bits  3, 1
```

.bss *Reserve Space in the .bss Section*

Syntax

.bss *symbol*,*size in words*[, *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.

This directive is similar to the .usect directive (see .usect topic); 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.

NOTE: This directive is supported only in EABI mode.

- The *symbol* is a required parameter. It defines a symbol 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 words* is a required parameter; it must be an absolute constant expression. The assembler allocates size words in the .bss section. There is no default size.
- The *blocking flag* is an optional parameter. If you specify a value greater than 0 for this parameter, the assembler allocates size in words contiguously. This means that the allocated space does not cross a page boundary unless its size is greater than a page, in which case the object starts on a page boundary.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. The boundary must be set to a power of 2 between 2^0 and 2^{15}, inclusive. If the SPC is already aligned at the specified boundary, it is not incremented.

For more information about sections, see Chapter 2.

.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 values into consecutive words of the current section. Each byte is placed in a word by itself; the eight MSBs are filled with 0s. A *value* can be one of the following:

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

Values are not packed or sign-extended; each byte occupies the eight least significant bits of a full 16-bit word. The assembler truncates values greater than eight 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. The label STRX has the value 100h, which is the location of the first initialized word.

```
1 000000               .space   100h * 16
2 000100 000A STRX     .byte    10, -1, "abc", 'a'
  000101 00FF
  000102 0061
  000103 0062
  000104 0063
  000105 0061
3 000106 000A          .char    10, -1, "abc", 'a'
  000107 00FF
  000108 0061
  000109 0062
  00010a 0063
  00010b 0061
```

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

Syntax **Single Line:**

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

Syntax **Multiple Lines:**

.cdecls [*options*]

%{

/*----------*/

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

/*----------*/

%}

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

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

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 #defines.

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

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

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

See Chapter 13 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 "1",_OPTIMIZE_FOR_SPACE
A     7                          .define "1",__ASM_HEADER__
A     8                          .define "1",__edg_front_end__
A     9                          .define "5001000",__COMPILER_VERSION__
A    10                          .define "0",__TI_STRICT_ANSI_MODE__
A    11                          .define """14:53:42""",__TIME__
A    12                          .define """I""",__TI_COMPILER_VERSION_QUAL__
A    13                          .define "unsigned long",__SIZE_T_TYPE__
A    14                          .define "long",__PTRDIFF_T_TYPE__
A    15                          .define "1",__TMS320C2000__
A    16                          .define "1",_TMS320C28X
A    17                          .define "1",_TMS320C2000
A    18                          .define "1",__TMS320C28X__
A    19                          .define "1",__STDC__
A    20                          .define "1",__signed_chars__
A    21                          .define "0",__GNUC_MINOR__
```

```
A    22                         .define "1",_TMS320C28XX
A    23                         .define "5001000",__TI_COMPILER_VERSION__
A    24                         .define "1",__TMS320C28XX__
A    25                         .define "1",__little_endian__
A    26                         .define "199409L",__STDC_VERSION__
A    27                         .define """EDG gcc 3.0 mode""",__VERSION__
A    28                         .define """John\n""",NAME
A    29                         .define "unsigned int",__WCHAR_T_TYPE__
A    30                         .define "1",__TI_RUNTIME_RTS__
A    31                         .define "3",__GNUC__
A    32                         .define "10",WANT_ID
A    33                         .define """Sep  7 2007""",__DATE__
A    34                         .define "7250",__TI_COMPILER_VERSION_QUAL_ID__
A    35
A    36              ; =========== TYPE DEFINITIONS ===========
A    37              status_enum   .enum
A    38         0001  OK           .emember 1
A    39         0100  FAILED       .emember 256
A    40         0000  RUNNING      .emember 0
A    41                           .endenum
A    42
A    43              myCstruct     .struct 0,2 ; struct size=(4 bytes|64 bits), alignment=2
A    44         0000  member_a      .field 16   ; int member_a - offset 0 bytes, size (1 bytes|16 bits)
A    45         0001                .field 16   ; padding
A    46         0002  member_b      .field 32   ; float member_b-offset 2 bytes, size (2 bytes|32 bits)
A    47         0004                .endstruct  ; final size=(4 bytes|64 bits)
A    48
A    49              ; =========== EXTERNAL FUNCTIONS ===========
A    50                              .global _cvt_integer
A    51
A    52              ; =========== EXTERNAL VARIABLES ===========
A    53                              .global _a_variable
      2 00000000 0004  size: .int $sizeof(myCstruct)
      3 00000001 0000  aoffset: .int myCstruct.member_a
      4 00000002 0002  boffset: .int myCstruct.member_b
      5 00000003 0001  okvalue: .int status_enum.OK
      6 00000004 0100  failval: .int status_enum.FAILED
      7                       .if $defined(WANT_ID)
      8 00000005 004A  id .cstring NAME
        00000006 006F
        00000007 0068
        00000008 006E
        00000009 000A
        0000000a 0000
      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.

NOTE: The .clink directive is supported only for the COFF ABI. It is ignored when used in EABI mode.

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.

The .clink directive is useful only with the COFF object file format. Under the COFF ABI model, the linker assumes that all sections are ineligible for removal via conditional linking by default. If you want to make a section eligible for removal, you must apply a .clink directive to it. In contrast, under the ELF EABI model, the linker assumes that all sections are eligible for removal via conditional linking. Therefore, the .clink directive has no effect under EABI.

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:       .long   01Ah
   000001 0000
 6 000002 001A  Y:       .word   01Ah
 7 000003 001A  Z:       .word   01Ah
 8               ; Counts section is conditionally linked
 9                       .clink
10
11 000004 001A  XCount: .word   01Ah
12 000005 001A  YCount: .word   01Ah
13 000006 001A  ZCount: .word   01Ah
14              ; By default, .text in unconditionally linked
15 000000               .text
16
17 000000 97C6          MOV     *XAR6, AH
18              ; These references to symbol X cause the Vars
19              ; section to be linked into the COFF output
20 000001 8500+         MOV     ACC, @X
21 000002 3100          MOV     P, #0
22 000003 0FAB          CMPL    ACC, P
```

.common *Create a Common Symbol*

Syntax

.common *symbol*,*size in bytes*[, *alignment*]

.common *symbol*,*structure tag*[, *alignment*]

Description

The **.common** directive creates a common symbol in a common block, rather than placing the variable in a memory section.

NOTE: This directive is supported only when using EABI mode.

The benefit of common symbols is that generated code can remove unused variables that would otherwise increase the size of the .bss section. (Uninitialized variables of a size larger than 32 bytes are separately optimized through placement in separate subsections that can be omitted from a link.)

- The *symbol* is a required parameter. It defines a name for the symbol created by this 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 section used for common symbols. There is no default size.
- A *structure tag* can be used in place of a size to specify a structure created with the .struct directive. Either a size or a structure tag is required for this argument.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. The boundary must be set to a power of 2 between 2^0 and 2^{15}, inclusive. If the SPC is already aligned at the specified boundary, it is not incremented.

Common symbols are symbols that are placed in the symbol table of an ELF object file. They represent an uninitialized variable. Common symbols do not reference a section. (In contrast, initialized variables need to reference a section that contains the initialized data.) The value of a common symbol is its required alignment; it has no address and stores no address. While symbols for an uninitialized common block can appear in executable object files, common symbols may only appear in relocatable object files. Common symbols are preferred over weak symbols. See the section on the "Symbol Table" in the System V ABI specification for more about common symbols.

When object files containing common symbols are linked, space is reserved in an uninitialized section (.common) for each common symbol. A symbol is created in place of the common symbol to refer to its reserved location.

.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 C2000_A_DIR environment variable
4. Any directories specified by the C2000_C_DIR environment variable

For more information about the --include_path option and C2000_A_DIR, see Section 4.5. For more information about C2000_C_DIR, see the *TMS320C28x 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)

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

Listing file:

```
1 000000           .space 29
2                  .copy "byte.asm"
1               ** In byte.asm
2 000002 0005      byte 5
3                  .copy "word.asm"
1               ** In word.asm
2 000003 ABCD      .word 0ABCDh
4               * Back in byte.asm
5 000004 0006      .byte 6
3
4               **Back in original file
5 000005 646F      .string "done"
  000006 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)

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

Listing file:

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

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

Example

This example illustrates a structure in C that will be accessed in assembly code.

```
;typedef struct MYSTR1
;{  long  l0;       /* offset 0 */
;   short s0;       /* offset 2 */
;}  MYSTR1;         /* size 4, alignment 2 */
;typedef struct MYSTR2
```

```
;{  MYSTR1 m1;      /* offset 0 */
;   short s1;       /* offset 4 */
;}  MYSTR2;         /* size 6, alignment 2 */
;
; The structure will get the following offsets once the C compiler lays out the structure
; elements according to C standard rules:
;
; offsetof(MYSTR1, l0) = 0
; offsetof(MYSTR1, s0) = 2
; sizeof(MYSTR1)       = 4
;
; offsetof(MYSTR2, m1) = 0
; offsetof(MYSTR2, s1) = 4
; sizeof(MYSTR2)       = 6
;
; Attempts to replicate this structure in assembly using .struct/.union directives will not
; create the correct offsets because the assembler tries to use the most compact arrangement:

MYSTR1        .struct
l0        .long              ; bytes 0 and 1
s0        .short           ; byte 2
M1_LEN          .endstruct         ; size 4, alignment 2

MYSTR2        .struct
m1        .tag MYSTR1        ; bytes 0-3
s1        .short           ; byte 4
M2_LEN          .endstruct         ; size 6, alignment 2

                .sect   "data1"
                .word   MYSTR1.l0
                .word   MYSTR1.s0
                .word   M1_LEN

                .sect   "data2"
                .word   MYSTR2.m1
                .word   MYSTR2.s1
                .word   M2_LEN

; The .cstruct/.cunion directives calculate offsets the same as the C compiler. The resulting
; assembly structure can be used to access elements of the C structure. Compare differences
; in the offsets of those structures defined via .struct above and the offsets for C code.

CMYSTR1           .cstruct
l0              .long
s0              .short
MC1_LEN           .endstruct

CMYSTR2           .cstruct
m1              .tag CMYSTR1
s1              .short
MC2_LEN           .endstruct

                .sect   "data3"
                .word   CMYSTR1.l0, MYSTR1.l0
                .word   CMYSTR1.s0, MYSTR1.s0
                .word   MC1_LEN, M1_LEN

                .sect   "data4"
                .word   CMYSTR2.m1, MYSTR2.m1
                .word   CMYSTR2.s1, MYSTR2.s1
                .word   MC2_LEN, M2_LEN
```

.data *Assemble Into the .data Section*

Syntax

.data

Description

The **.data** directive sets .data as the current section; the lines that follow will be assembled into the .data 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 and .text sections.

```
      1              ****************************************
      2              **     Reserve space in .data.        **
      3              ****************************************
      4 000000               .data
      5 000000               .space          0CCh
      6              ****************************************
      7              **    Assemble into .text.            **
      8              ****************************************
      9 000000               .text
     10        0000 INDEX  .set            0
     11 000000 9A00          MOV            AL,#INDEX
     12             ****************************************
     13             **  Assemble into .data.              **
     14             ****************************************
     15 00000c       Table: .data
     16 00000d FFFF         .word   -1   ; Assemble 16-
bit constant into .data.
     17 00000e 00FF         .byte   0FFh ; Assemble 8-
bit constant into .data.
     18             ****************************************
     19             **  Assemble into .text.              **
     20             ****************************************
     21 000001              .text
     22 000001 08A9"        ADD            AL,Table
        000002 000C
     23             ****************************************
     24             **  Resume assembling into the .data  **
     25             **  section at address 0Fh.           **
     26             ****************************************
     27 00000f              .data
```

.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
       7
       9                .loop   3
      10                .eval   x+1, x
      11                .endloop
```

.elfsym *ELF Symbol Information*

Syntax

.elfsym *name*, **SYM_SIZE(***size***)**

Description

The .elfsym directive provides additional information for symbols in the ELF format. This directive is designed to convey different types of information, so the *type*, *data* pair is used to represent each type. Currently, this directive only supports the SYM_SIZE type.

NOTE: This directive is supported for EABI mode only.

SYM_SIZE indicates the allocation size (in bytes) of the symbol indicated by *name*.

Example

This example shows the use of the ELF symbol information directive.

```
            .sect      ".examp"
            .align 4
            .elfsym    ex_sym, SYM_SIZE(4)
ex_sym:
             .word     0
```

.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

This example sends the message ERROR -- MISSING PARAMETER to the standard output device.

Source file:

```
         .global     PARAM
MSG_EX   .macro parm1
         .if    $symlen(parm1) = 0
         .emsg  "ERROR -- MISSING PARAMETER"
         .else
          ADD   AL, @parm1
         .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     AL, @parm1
       7                    .endif
       8                    .endm
       9
      10 000000             MSG_EX PARAM
1                           .if     $symlen(parm1) = 0
1                           .emsg   "ERROR -- MISSING PARAMETER"
1                           .else
1         000000 9400!      ADD     AL, @PARAM
1                           .endif
      11
      12 000001             MSG_EX
1                          .if     $symlen(parm1) = 0
1                          .emsg   "ERROR -- MISSING PARAMETER"
  ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
1                           .else
1                           ADD     AL, @parm1
1                          .endif
```

```
 1 Error, No Warnings
```

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

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

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

.end *End Assembly*

Syntax

.end

Description

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

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

Ending a Macro

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

Example

This example shows how the .end directive terminates assembly. Any source statements that follow the .end directive are ignored by the assembler.

Source file:

```
START:  .space  300
TEMP    .set    15
LOC1    .usect  ".ebss", 48h
        ABS     ACC
        ADD     ACC, #TEMP
        MOV     @LOC1, ACC
        .end
        .byte   4
        .word   CCCh
```

Listing file:

```
      1 000000       START:  .space  300
      2        000F  TEMP    .set    15
      3 000000       LOC1    .usect  ".ebss", 48h
      4 000013 FF56          ABS     ACC
      5 000014 090F          ADD     ACC, #TEMP
      6 000015 9600-         MOV     @LOC1, ACC
      7                      .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   ACC, #1024
        .else
        ADD   ACC, #1024*4
        .endif
        .fcnolist
        .if    AAA
        ADD   ACC, #1024
        .else
        ADD   ACC, #1024*10
        .endif
```

Listing file:

```
       1        0001  AAA    .set  1
       2        0000  BBB    .set  0
       3                     .fclist
       4
       5                     .if   AAA
       6 000000 FF10         ADD   ACC, #1024
         000001 0400
       7                     .else
       8                     ADD   ACC, #1024*4
       9                     .endif
      10
      11                     .fcnolist
      12
      14 000002 FF10         ADD   ACC, #1024
         000003 0400
```

.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. The default size is 16 bits. If you specify a size in bits of 16 or more, the field starts on a word boundary. If you specify a value that cannot fit in *size in bits*, the assembler truncates the value and issues a warning message. For example, .field 3,1 causes the assembler to truncate the value 3 to 1; the assembler also prints the message:

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

Successive .field directives pack values into the specified number of bits starting at the current word. Fields are packed starting at the least significant part of the word, moving toward the most significant part as more fields are added. If the assembler encounters a field size that does not fit into the current word, it writes out the word, increments the SPC, and begins packing fields into the next word. You can use the .align directive with an operand of 1 to force the next .field directive to begin packing into a new word.

The .field directive is similar to the .bits directive (see the .bits topic). However, the .bits directive does not force alignment to a field boundary and does not automatically increment the SPC when a word boundary is reached.

Use the .align directive to force the next .field directive to begin packing 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.

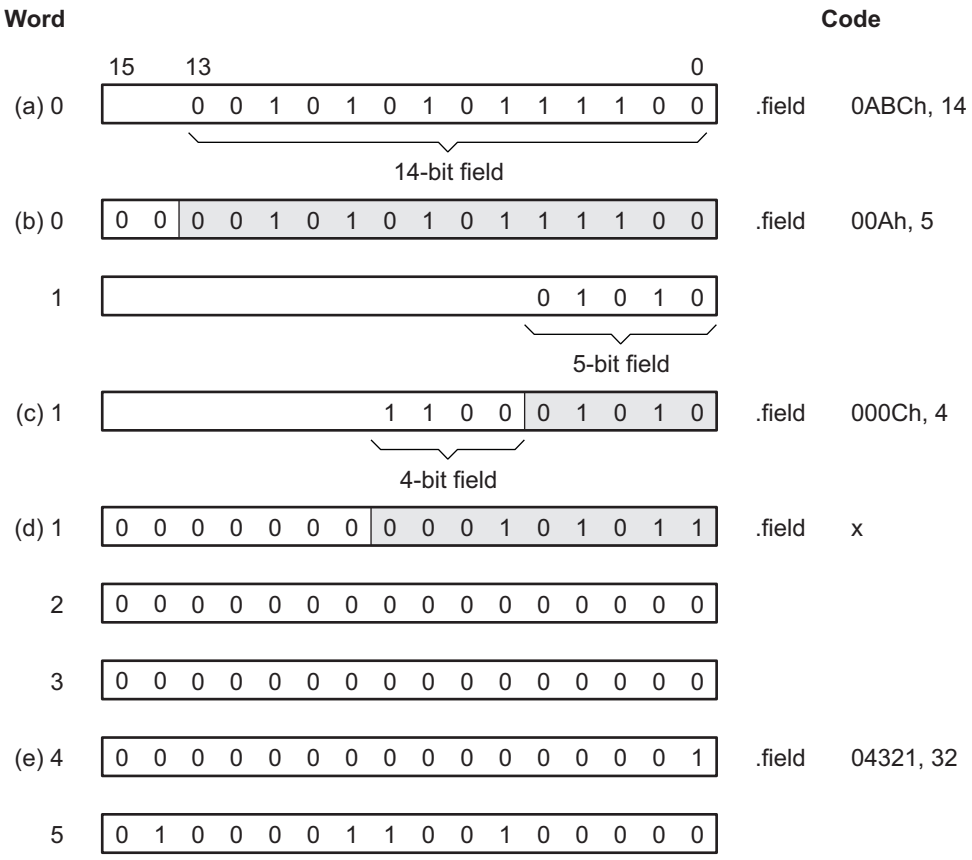
Example

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.

```
 1                ***********************************
 2                **    Initialize a 14-bit field.  **
 3                ***********************************
 4 000000 0ABC           .field  0ABCh, 14
 5
 6                ***********************************
 7                **    Initialize a 5-bit field    **
 8                **         in a new word.         **
 9                ***********************************
10 000001 000A   L_F:    .field  0Ah, 5
11
12                ***********************************
13                **    Initialize a 4-bit field    **
14                **       in the same word.        **
15                ***********************************
16 000001 018A   X:      .field  0Ch, 4
17                ***********************************
18                **    Relocatable field           **
19                **        in the next 2 words.    **
20                ***********************************
21 000002 0001'          .field  X
22                ***********************************
23                **    Initialize a 32-bit field   **
24                ***********************************
25 000003 4321           .field  04321h, 32
   000004 0000
```

Figure 5-5 shows how the directives in this example affect memory.

Figure 5-5. The .field Directive



.float/.xfloat/.xldouble *Initialize Single- and Double-Precision Floating-Point Values*

Syntax

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

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

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

Description

The **.float** and **.xfloat** directives place the IEEE single-precision floating-point representation of a single floating-point constant into a word in the current section. The *value* must be an absolute constant expression with an arithmetic type or a symbol equated to an absolute constant expression with an arithmetic type. Each constant is converted to a floating-point value in IEEE single-precision 32-bit format.

The .float directive aligns the floating-point constants on the long-word boundary, while the .xfloat directive does not.

The 32-bit value is stored exponent byte first, least significant word of fraction second, and most significant word of fraction third, in the format shown in Figure 5-6.

The **.xldouble** directive places the IEEE double-precision floating-point representation of a double floating-point constant into two words in the current section. The *value* must be an absolute constant expression with an arithmetic type or a symbol equated to an absolute constant expression with an arithmetic type. Each constant is converted to a floating-point double value in IEEE double-precision 64-bit format. The 64-bit value is stored exponent byte first, least significant word of fraction second, and most significant word of fraction third.

Figure 5-6. 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 in a .struct/.endstruct sequence, .float defines a member's size; it does not initialize memory. For more information, see the .struct/.endstruct/.tag topic.

Example

Following are examples of the .float and .xfloat directives:

```
1 00000000 5951          .float  -1.0e25
  00000001 E904
2 00000002 0010          .byte   0x10
3 00000003 0000          .xfloat 123.0   ; not on long-word boundary
  00000004 42F6
4 00000006 0000          .float  3       ; aligns on long-word boundary
  00000007 4040
```

.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. If a global symbol is defined more than once, the linker issues a multiple-definition error. (The assembler can provide a similar multiple-definition error for local symbols.) 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 0956         ADD    ACC, #56h
 7
 8 000001 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          0001  X:        .set    1
 6          0002  Y:        .set    2
 7          0003  Z:        .set    3
 8 000000 0000!             .word   INIT
 9                ;         .
10                ;         .
11                ;         .
12                          .end
```

file3.lst

```
 1                ; Global symbol defined in this file
 2                      .def    INIT
 3                ; Global symbols defined in file4.lst
 4                      .ref    X, Y, Z
 5 000000         INIT:
 6 000000 0956          ADD     ACC, #56h
 7
 8 000001 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          0001  X:        .set    1
 6          0002  Y:        .set    2
 7          0003  Z:        .set    3
 8 000000 0000!             .word   INIT
 9                ;         .
10                ;         .
11                ;         .
12                          .end
```

.group/.gmember/.endgroup *Define Common Data Section*

Syntax

.group *group section name group type*

.gmember *section name*

.endgroup

Description

Three directives instruct the assembler to make certain sections members of an ELF group section (see the ELF specification for more information on group sections).

NOTE: These directives are supported for EABI mode only.

The **.group** directive begins the group declaration. The *group section name* designates the name of the group section. The *group type* designates the type of the group. The following types are supported:

0x0	Regular ELF group
0x1	COMDAT ELF group

Duplicate COMDAT (common data) groups are allowed in multiple modules; the linker keeps only one. Creating such duplicate groups is useful for late instantiation of C++ templates and for providing debugging information.

The **.gmember** directive designates *section name* as a member of the group.

The **.endgroup** directive ends the group declaration.

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

Syntax

.if *condition*

[**.elseif** *condition*]

[**.else**]

.endif

Description

These directives provide conditional assembly:

The **.if** directive marks the beginning of a conditional block. The *condition* 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 is optional in a conditional block, and more than one .elseif can be used. If an expression is false and there is no .elseif, 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 .elseif and .else directives can be used in the same conditional assembly block.

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

See Section 4.9.2 for information about relational operators.

Example

This example shows conditional assembly:

```
 1        0001  SYM1   .set    1
 2        0002  SYM2   .set    2
 3        0003  SYM3   .set    3
 4        0004  SYM4   .set    4
 5
 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
12              If_5:  .if     SYM1 <= 10
13 000001 000A         .byte   10       ; Less than / equal
14                     .else
15                     .byte   SYM1     ; Greater than
16                     .endif
17
18              If_6:  .if     SYM3 * SYM2 != SYM4 + SYM2
19                     .byte   SYM3 * SYM2  ; Unequal value
20                     .else
21 000002 0008         .byte   SYM4 + SYM4  ; Equal values
22                     .endif
23
24              If_7:  .if SYM1 = 2
25                     .byte   SYM1
26                     .elseif SYM2 + SYM3 = 5
27 000003 0005         .byte   SYM2 + SYM3
28                     .endif
```

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

Syntax

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

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

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

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

Description

The **.int**, **.unint**, **.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.

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

This example uses the .int directive to initialize words.

```
1 000000                .space 73h
2 000000       PAGE     .usect ".ebss", 128
3 000080       SUMPTR   .usect ".ebss", 3
4 000008 FF20  INST:    MOV     ACC, #056h
  000009 0056
5 00000a 000A         .int 10, SYMPTR, -1, 35 + 'a', INST
  00000b 0080-
  00000c FFFF
  00000d 0084
  00000e 0008'
```

Example 2

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

```
1 000000 0C80  WORDX: .word  3200, 1 + 'AB', -0AFh, 'X'
  000001 4242
  000002 FF51
  000003 0058
```

.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. See Section 3.5 for more information about run-time relocation.

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 8.5.6 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 4.3), the assembler ignores the .list directive.

Example

This example shows how the .copy directive inserts source statements from another file. The first time .copy is encountered, the assembler lists the copied source lines in the listing file. The second time .copy 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. Also the line counter is incremented, even when source statements are not listed.

Source file:

copy.asm (source file)	copy2.asm (copy file)

`    .copy "copy2.asm"`
`** Back in original file`
`    NOP`
`    .nolist`
`    .copy "copy2.asm"`
`    .list`
`** Back in original file`
`    .string "done"`
	`** In copy2.asm`
	`    .word 32, 1 + 'A'`

Listing file:

```
     1                .copy    "copy2.asm"
     1              *In copy2.asm (copy file)
     2 000000 0020     .word 32, 1 + 'A'
       000001 0042
     2              * Back in original file
     3 000002 7700     NOP
     7              * Back in original file
     8 000005 0044     .string  "Done"
       000006 006F
       000007 006E
       000008 0065
```

.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 directive aligns the result on the long-word boundary, while .xlong does not.

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 in a .struct/.endstruct sequence, .long defines a member's size; it does not initialize memory. See the .struct/.endstruct/.tag topic.

Example

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

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

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

Syntax

.loop [*count*]

.break [*end-condition*]

.endloop

Description

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

The **.loop** directive begins a repeatable block of code. The optional *count* operand, if used, must be a well-defined integer expression. The *count* indicates the number of loops to be performed (the loop count). If *count* is omitted, it defaults to 1024. The loop will be repeated count number of times, unless terminated early by a .break directive.

The optional **.break** directive terminates a .loop early. You may use .loop without using .break. The .break directive terminates a .loop only if the *end-condition* expression is true (evaluates to nonzero). If the optional *end-condition* operand is omitted, it defaults to true. If *end-condition* is true, the assembler stops repeating the .loop body immediately; any remaining statements after .break and before .endloop are not assembled. The assembler resumes assembling with the statement after the .endloop directive. If *end-condition* is false (evaluates to 0), the loop continues.

The **.endloop** directive marks the end of a repeatable block of code. When the loop terminates, whether by a .break directive with a true *end-condition* or by performing the loop count number of iterations, the assembler stops repeating the loop body and resumes assembling with the statement after the .endloop directive.

Example

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

```
      1                    .eval       0,x
      2              COEF  .loop
      3                    .word       x*100
      4                    .eval       x+1, x
      5                    .break      x = 6
      6                    .endloop
1       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 6.

.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 C2000_A_DIR environment variable
4. Any directories specified by the C2000_C_DIR environment variable

See Section 4.5 for more information about the --include_path option.

A .mlib directive causes the assembler to open the library specified by *filename* and create a table of the library's contents. The assembler stores names of individual library members in the opcode table as library entries. This redefines any existing opcodes or macros with the same name. If one of these macros is called, the assembler extracts the library entry and loads it into the macro table. The assembler expands the library entry as it does other macros, but it does not place the source code in the listing. Only macros called from the library are extracted, and they are extracted only once.

See Chapter 6 for more information on macros and macro libraries.

Example

The code creates a macro library that defines two macros, inc1.asm and dec1.asm. The file inc1.asm contains the definition of inc1 and dec1.asm contains the definition of dec1.

inc1.asm

```
* Macro for incrementing
 inc1 .macro A
     ADD A, #1
     .endm
```

dec1.asm

```
* Macro for decrementing
 dec1 .macro A
     SUB A, #1
     .endm
```

Use the archiver to create a macro library:

```
ar2000 -a mac inc1.asm dec1.asm
```

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

```
       1                .mlib    "mac.lib"
       2
       3        * Macro call
       4 000000         inc1    AL
1        000000 9C01    ADD     AL,#1
       5
       6        * Macro call
       7 000001         dec1    AR1
1        000001 08A9    SUB     AR1,#1
         000002 FFFF
```

.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 6 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                     .string ":p1:", ":p2:", ":p3:"
      3                     .endm
      4
      5 000000               STR_3 "as", "I", "am"
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
      6 00000c 003A          .string ":p1:", ":p2:", ":p3:"
        00000d 0070
        00000e 0031
        00000f 003A
        000010 003A
        000011 0070
        000012 0032
        000013 003A
        000014 003A
        000015 0070
        000016 0033
        000017 003A
      7
      8                     .mnolist
      9 000018               STR_3 "as", "I", "am"
     10                     .mlist
     11 000024               STR_3 "as", "I", "am"
1       000024 003A          .string ":p1:", ":p2:", ":p3:"
        000025 0070
        000026 0031
        000027 003A
        000028 003A
        000029 0070
        00002a 0032
```

```
                    00002b 003A
                    00002c 003A
                    00002d 0070
                    00002e 0033
                    00002f 003A
12                  000030 003A         .string ":p1:", ":p2:", ":p3:"
                    000031 0070
                    000032 0031
                    000033 003A
                    000034 003A
                    000035 0070
                    000036 0032
                    000037 003A
                    000038 003A
                    000039 0070
                    00003a 0033
                    00003b 003A
13
```

.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, and cannot be used in expressions. 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 4.8.3 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                 0076  B  .set    76h
 3
 4 00000000 F800!           MOV     DP, #ADDRA
 5
 6 00000001 8500! LABEL1:   MOV     ACC, @ADDRA
 7 00000002 1976            SUB     ACC, #B
 8 00000003 6403            B       $1, LT
 9 00000004 9600!           MOV     @ADDRB, ACC
10 00000005 6F02            B       $2, UNC
11
12 00000006 8500! $1       MOV     ACC, @ADDRA
13 00000007 8100! $2       ADD     ACC, @ADDRC
14                         .newblock       ; Undefine $1 to use again.
15
16 00000008 6402            B       $1, LT
17 00000009 9600!           MOV     @ADDRC, ACC
18 0000000a 7700  $1        NOP
```

.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).
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, L, M, T, and W (turns off the limits of B, L, M, T, and W).
T	limits the listing of .string directives to one line.
W	limits the listing of .word and .int directives to one line.
X	produces a cross-reference listing of symbols. You can also obtain a cross-reference listing by invoking the assembler with the --asm_listing_cross_reference option (see Section 4.3).

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 to 1 **
 4              **         to 1 line each.            **
 5              ****************************************
 6                   .option B, W, L, T
 7 000000 00BD       .byte   -'C', 0B0h, 5
 8 000004 CCDD       .long   0AABBCCDDh, 536 + 'A'
 9 000008 15AA       .word   5546, 78h
10 00000a 0045       .string "Extended Registers"
11              ****************************************
12              **     Reset the listing options.     **
13              ****************************************
14                   .option R
15 00001c 00BD       .byte   -'C', 0B0h, 5
   00001d 00B0
   00001e 0005
16 000020 CCDD       .long   0AABBCCDDh, 536 + 'A'
   000021 AABB
   000022 0259
   000023 0000
17 000024 15AA       .word   5546, 78h
   000025 0078
18 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:

```
TMS320C000 COFF Assembler       Version x.xx        Day    Time    Year
Copyright (c) 1996-2011    Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE    1

       2                    ;        .
       3                    ;        .
       4                    ;        .
TMS320C2000 COFF Assembler       Version x.xx        Day      Time      Year
Copyright (c) 1996-2011    Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE    2


 No Errors, No Warnings
```

.retain / .retainrefs *Conditionally Retain Sections In Object Module Output*

Syntax

.retain["*section name*"]

.retainrefs["*section name*"]

Description

The **.retain** directive indicates that the current or specified section is not eligible for removal via conditional linking. You can also override conditional linking for a given section with the --retain linker option. You can disable conditional linking entirely with the --unused_section_elimination=off linker option.

The **.retainrefs** directive indicates that any sections that refer to the current or specified section are not eligible for removal via conditional linking. For example, applications may use an .intvecs section to set up interrupt vectors. The .intvecs section is eligible for removal during conditional linking by default. You can force the .intvecs section and any sections that reference it to be retained by applying the .retain and .retainrefs directives to the .intvecs section.

NOTE: The .retain and .retainrefs directives are supported only for EABI. They are ignored when used with the COFF ABI.

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

The linker assumes that all sections by default are eligible for removal via conditional linking. (However, the linker does automatically retain the .reset section.) The .retain directive is useful for overriding this default conditional linking behavior for sections that you want to keep included in the link, even if the section is not referenced by any other section in the link. For example, you could apply a .retain directive to an interrupt function that you have written in assembly language, but which is not referenced from any normal entry point in the application.

Under the COFF ABI model, the linker assumes that all sections are not eligible for removal via conditional linking by default. So under the COFF ABI mode, the .retain directive does not have any real effect on the section.

.sblock *Specify Blocking for a 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. 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. 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* **"**

.sect " *section name* **"** [,{RO|RW}] [,{ALLOC|NOALLOC}]

Description

The **.sect** directive defines a named section that can be used like the default .text and .data sections. The .sect directive sets *section name* to be the current section; the lines that follow are assembled into the *section name* 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.

If you are using EABI, the sections can be marked read-only (RO) or read-write (RW). Also, the sections can be marked for allocation (ALLOC) or no allocation (NOALLOC). These attributes can be specified in any order, but only one attribute from each set can be selected. RO conflicts with RW, and ALLOC conflicts with NOALLOC. If conflicting attributes are specified the assembler generates an error, for example:

```
"t.asm", ERROR!  at line 1:[E0000] Attribute RO cannot be combined with attr RW
          .sect "illegal_sect",RO,RW
```

Example

This example defines two special-purpose sections, Sym_Defs and Vars, and assembles code into them.

```
 1                    **   Begin assembling into .text section.   **
 2 000000                       .text
 3 000000 FF20                  MOV     ACC, #78h  ; Assembled into .text
   000001 0078
 4 000002 0936                  ADD     ACC, #36h  ; Assembled into .text
 5
 6                    **  Begin assembling into Sym_Defs section. **
 7 000000                       .sect   "Sym_Defs"
 8 000000 CCCD                  .float  0.         ; Assembled into Sym_Defs
   000001 3D4C
 9 000002 00AA        X:        .word   0AAh       ; Assembled into Sym_Defs
10 000003 FF10                  ADD     ACC, #X    ; Assembled into Sym_Defs
   000004 0002+
11
12                    **   Begin assembling into Vars section. **
13 000000                       .sect   "Vars"
14        0010        WORD_LEN  .set    16
15        0020        DWORD_LEN .set    WORD_LEN * 2
16        0008        BYTE_LEN  .set    WORD_LEN / 2
17        0053        STR       .set    53h
18
19                    **   Resume assembling into .text section.  **
20 000003                       .text
21 000003 0942                  ADD     ACC, #42h  ; Assembled into .text
22 000004 0003                  .byte   3, 4       ; Assembled into .text
   000005 0004
23
24                    **   Resume assembling into Vars section. **
25 000000                       .sect   "Vars"
26 000000 000D                  .field  13, WORD_LEN
27 000001 000A                  .field  0Ah, BYTE_LEN
28 000002 0008                  .field  10q, DWORD_LEN
   000003 0000
29
```

.set — *Define Assembly-Time Constant*

Syntax

symbol **.set** *value*

Description

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

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

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

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

Symbols defined with .set can be made externally visible with the .def or .global directive (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.

```
 1                ********************************************
 2                **   Equate symbol AUX_R1 to register AR1   **
 3                **   and use it instead of the register.    **
 4                ********************************************
 5         0001   AUX_R1  .set    AR1
 6  000000 28C1           MOV     *AUX_R1, #56h
    000001 0056
 7
 8                ********************************************
 9                **   Set symbol index to an integer expr.   **
10                **    and use it as an immediate operand.   **
11                ********************************************
12         0035   INDEX   .set    100/2 +3
13  000002 0935           ADD     ACC, #INDEX
14
15                ********************************************
16                ** Set symbol SYMTAB to a relocatable expr. **
17                **   and use it as a relocatable operand.   **
18                ********************************************
19  000003 000A   LABEL   .word   10
20         0004'  SYMTAB  .set    LABEL + 1
21
22                ********************************************
23                **   Set symbol NSYMS equal to the symbol   **
24                **   INDEX and use it as you would INDEX.   **
25                ********************************************
26         0035   NSYMS   .set    INDEX
27  000004 0035           .word   NSYMS
```

.space/.bes *Reserve Space*

Syntax

[*label*] **.space** *size in bits*

[*label*] **.bes** *size in bits*

Description

The **.space** and **.bes** directives reserve 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. When you use a label with the .bes directive, it points to the *last* reserved.

Example

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

```
 1                ******************************************
 2                **  Begin assembling into .text section.   **
 3                ******************************************
 4  000000                .text
 5                ******************************************
 6                **  Reserve 0F0 bits (15 words in the      **
 7                **            .text section.               **
 8                ******************************************
 9  000000                .space  0F0h
10  00000f 0100           .word   100h, 200h
    000010 0200
11                ******************************************
12                **  Begin assembling into .data section.   **
13                ******************************************
14  000000                .data
15  000000 0049           .string "In .data"
    000001 006E
    000002 0020
    000003 002E
    000004 0064
    000005 0061
    000006 0074
    000007 0061
16                ******************************************
17                **  Reserve 100 bits in the .data section; **
18                **   RES_1 points to the first word that   **
19                **         contains reserved bits.         **
20                ******************************************
21  000008        RES_1: .space  100
22  00000f 000F           .word   15
23                ******************************************
24                **  Reserve 20 bits in the .data section;  **
25                **  RES_2 points to the last word that     **
26                **         contains reserved bits.         **
27                ******************************************
28  000011        RES_2: .bes    20
29  000012 0036           .word   36h
30  000013 0011"          .word   RES_
```

.sslist/.ssnolist *Control Listing of Substitution Symbols*

Syntax

.sslist

.ssnolist

Description

Two directives allow you to control substitution symbol expansion in the listing file:

The **.sslist** directive allows substitution symbol expansion in the listing file. The expanded line appears below the actual source line.

The **.ssnolist** directive suppresses substitution symbol expansion in the listing file.

By default, all substitution symbol expansion in the listing file is suppressed; the assembler acts as if the .ssnolist directive had been used.

Lines with the pound (#) character denote expanded substitution symbols.

Example

This example shows code that, by default, suppresses the listing of substitution symbol expansion, and it shows the .sslist directive assembled, instructing the assembler to list substitution symbol code expansion.

```
      1 00000000     ADDRX   .usect  ".ebss", 1
      2 00000001     ADDRY   .usect  ".ebss", 1
      3 00000002     ADDRA   .usect  ".ebss", 1
      4 00000003     ADDRB   .usect  ".ebss", 1
      5
      6              ADD2    .macro  parm1, parm2
      7                      MOV     ACC, @parm1
      8                      ADD     ACC, @parm2
      9                      MOV     @parm2, ACC
     10                      .endm
     11
     12 00000000               ADD2    ADDRX, ADDRY
1       00000000 8500-         MOV     ACC, @ADDRX
1       00000001 8101-         ADD     ACC, @ADDRY
1       00000002 9601-         MOV     @ADDRY, ACC
     13
     14                      .sslist
     15 00000003               ADD2    ADDRA, ADDRB
1       00000003 8502-         MOV     ACC, @parm1
#                      MOV     ACC, @ADDRA
1       00000004 8103-         ADD     ACC, @parm2
#                      ADD     ACC, @ADDRB
1       00000005 9603-         MOV     @parm2, AC
```

.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. With the .string directive, each 8-bit character has its own 16-bit word, but with the .pstring directive, the data is packed so that each word contains two 8-bit bytes. The *expr* or *string* can be one of the following:

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

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 words in the current section.

```
1 000000 0041  Str_Ptr:  .string  "ABCD"
  000001 0042
  000002 0043
  000003 0044
2
3 000004 0041            .string  41h, 42h, 43h, 44h
  000005 0042
  000006 0043
  000007 0044
4
5 000008 4175            .pstring "Austin", "Houston"
  000009 7374
  00000a 696E
  00000b 486F
  00000c 7573
  00000d 746F
  00000e 6E00
6
7 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. The 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, .int, .long, .word, .string, .pstring, .float, .field, 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

```
REAL_REC   .struct                  ; stag
NOM        .int                     ; member1 = 0
DEN        .int                     ; member2 = 1

REAL_LEN   .endstruct               ; real_len = 4
            ADD   ACC, @(REAL + REAL_REC.DEN) ;access structure element
REAL       .usect ".ebss", REAL_LEN           ; allocate mem rec
```

Example 2

```
CPLX_REC  .struct
REALI     .tag REAL_REC             ; stag
IMAGI     .tag REAL_REC             ; member1 = 0
CPLX_LEN  .endstruct                ; rec_len = 4

COMPLEX   .tag CPLX_REC             ; assign structure attrib
          ADD  ACC, COMPLEX.REALI   ; access structure
          ADD  ACC, COMPLEX.IMAGI
COMPLEX   .usect ".ebss", CPLX_LEN    ; allocate space
```

Example 3

```
               .struct         ; no stag puts mems into
     X         .int            ; global symbol table
     Y         .int            ;create 3 dim templates
     Z         .int
               .endstruct
```

Example 4

```
BIT_REC   .struct                   ; stag
STREAM    .string 64
BIT7      .field  7                 ; bits1 = 64
BIT9      .field  9                 ; bits2 = 64
BIT10     .field  10                ; bits3 = 65
X_INT     .int                      ; x_int = 67
BIT_LEN   .endstruct                ; length = 68

BITS      .tag   BIT_REC
          ADD    AC, @BITS.BIT7       ; move into acc
          AND    ACC, #007Fh          ; mask off garbage bits
BITS      .usect ".ebss", BIT_REC
```

.symdepend *Create an Artificial Reference from a Section to a Symbol*

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. If a global symbol is defined more than once, the linker issues a multiple-definition error. (The assembler can provide a similar multiple-definition error for local symbols.)

The .symdepend directive creates a symbol table entry only if the module actually uses the symbol. The .weak directive, in contrast, always creates a symbol table entry for a symbol, whether the module uses the symbol or not (see .weak topic).

If the symbol is *defined in the current module*, use the .symdepend directive to declare 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 7700          NOP
       3 000001 7700          NOP
       4 000002 7700          NOP
       5
       7 000003 7700      NOP
       8 000004 7700      NOP
       9 000005 7700      NOP
      10
      12 000006 7700                  NOP
      13 000007 7700                  NOP
      14 000008 7700                  NOP
```

.text *Assemble Into the .text Section*

Syntax

.text

Description

The **.text** sets .text as the current section. Lines that follow this directive will be assembled 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 character strings.

```
 1            ****************************************
 2            ** Begin assembling into .data section. **
 3            ****************************************
 4 000000                .data
 5 000000 000A           .byte   0Ah, 0Bh
   000001 000B
 6
 7            ****************************************
 8            ** Begin assembling into .text section. **
 9            ****************************************
10 000000                .text
11 000000 0041  START:   .string "A", "B", "C"
   000001 0042
   000002 0043
12 000003 0058  END:     .string "X", "Y", "Z"
   000004 0059
   000005 005A
13
14 000006 8100'          ADD     ACC, @START
15 000007 8103'          ADD     ACC, @END
16
17           ****************************************
18           ** Resume assembling into .data section.**
19           ****************************************
20 000002                .data
21 000002 000C           .byte   0Ch, 0Dh
   000003 000D
22           ****************************************
23           ** Resume assembling into .text section.**
24           ****************************************
25 000008                .text
26 000008 0051           .string "Quit"
   000009 0075
   00000a 0069
   00000b 0074
```

.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:

```
TMS320C2000 COFF Assembler    Version x.xx          Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Fast Fourier Transforms ****                                  PAGE    1

       2                    ;        .
       3                    ;        .
       4                    ;        .
TMS320C2000 COFF Assembler    Version x.xx          Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Floating-Point Routines ****                                  PAGE    2


 No Errors, No Warnings
```

.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 4.8.8 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 13 for more information about using C/C++ headers in assembly source.

.union/.endunion/.tag *Declare Union Type*

Syntax

[*utag*] **.union** [*expr*]

[mem_0] *element* [$expr_0$]
[mem_1] *element* [$expr_1$]
. . .
. . .
. . .

[mem_n] **.tag** *utag* [$expr_n$]
. . .
. . .
. . .

[mem_N] *element* [$expr_N$]

[*size*] **.endunion**

label **.tag** *utag*

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, .xldouble, .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
2                           .global employid
3                 xample    .union            ; utag
4           0000  ival      .int              ; member1 = int
5           0000  fval      .float            ; member2 = float
6           0000  sval      .string           ; member3 = string
7           0002  real_len  .endunion
8
9  00000000       employid   .usect ".ebss", real_len  ; allocate memory
10
11                employid   .tag xample       ; name an instance
12
13 00000000 08A1-             ADD AR1, #employid.ival
   00000001 0000
```

Example 2

```
1
2                       .union          ; utag
3         0000  x       .long           ; member 1= long
4         0000  y       .float          ; member 2 = float
5         0000  z       .int            ; member 3 = int
6         0002  size_u  .endunion       ; size_u = 2
```

.usect — *Reserve Uninitialized Space*

Syntax

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

Description

The **.usect** directive reserves space for variables in an uninitialized, named section. This directive is similar to the .bss directive (see .bss topic); 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 words* is an expression that defines the number of words that are reserved in *section name*.
- The *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 that the allocated space does not cross a page boundary (64 words) unless its size is greater than a page, in which case the allocated space starts on a page boundary. By default, the compiler causes this flag to be set to 0 so that DP load optimization is used. The compiler provides the "blocked" and "noblocked" variable attributes for controlling blocking on a per-variable basis. For examples of DP load optimization, see the Tools Insider blog in TI's E2E community.
- The *alignment* is an optional parameter. It causes the assembler to allocate the specified size in words on long word boundaries. The resulting alignment will be on a boundary that is 2 to the power of the specified alignment parameter. For example, an alignment parameter of 5 gives an alignment of 2**5, which is 32 words.

Initialized sections directives (.text, .data, and .sect) tell the assembler to pause assembling into the current section and 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.

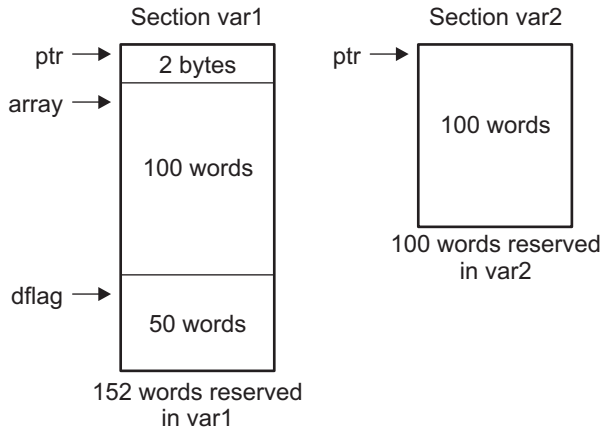
Example

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 words reserved in var1, and dflag points to the first word in a block of 50 words in var1. The symbol vec points to the first word reserved in the var2 section.

Figure 5-7 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 9A03       MOV     AL, #03h
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 000001 9C03       ADD     AL, #03h ; Still in .text
18
19          *****************************************
20          **    Reserve 50 words in var1.          **
21          *****************************************
22 000065    dflag  .usect  "var1", 50
23
24 000002 08A9       ADD     AL, #dflag  ; Still in .text
   000003 0065-
25
26          *****************************************
27          **    Reserve 100 words in var2.         **
28          *****************************************
29 000000     vec  .usect  "var2", 100
30
31 000004 08A9       ADD     AL, #vec  ; Still in .text
   000005 0000-
32
33          *****************************************
34          **    Declare an external .usect symbol  **
35          *****************************************
36                .global array
```

Figure 5-7. The .usect Directive



.var *Use Substitution Symbols as Local Variables*

Syntax

.var sym_1 [**,** sym_2 **,** ... **,** sym_n]

Description

The **.var** directive allows you to use substitution symbols as local variables within a macro. With this directive, you can define up to 32 local macro substitution symbols (including parameters) per macro.

The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

See Section 4.8.8 for more information on substitution symbols .See Chapter 6 for information on macros.

.weak *Identify a Symbol to be Treated as a Weak Symbol*

Syntax

.weak *symbol name*

Description

The **.weak** 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. Instead of including a weak symbol in the output file's symbol table by default (as it would for a global symbol), the linker only includes a weak symbol in the output of a "final" link if the symbol is required to resolve an otherwise unresolved reference. See Section 2.6.3 for details about how weak symbols are handled by the linker.

NOTE: The .weak directive is supported for EABI mode only.

The .weak directive is equivalent to the .ref directive, except that the reference has weak linkage.

The .weak directive always creates a symbol table entry for a symbol, whether the module uses the symbol or not. The .symdepend directive, in contrast, creates an symbol table entry only if the module actually uses the symbol (see .symdepend topic).

If a symbol is *not defined in the current module* (which includes macro, copy, and include files), use the .weak directive to tell 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.

For example, use the .weak and .set directives in combination as shown in the following example, which defines a weak absolute symbol "ext_addr_sym":

```
                .weak   ext_addr_sym
ext_addr_sym    .set    0x12345678
```

If you assemble such assembly source and include the resulting object file in the link, the "ext_addr_sym" in this example is available as a weak absolute symbol in a final link. It is a candidate for removal if the symbol is not referenced elsewhere in the application.

Macro Language Description

The TMS320C28x 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**

6.1 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 6.3 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 6.2, *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 6.4.

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 6.8.

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.

6.2 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 6.4.

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 6.9.

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 6.3.

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 6.7 for more information about macro comments.

Example 6-1 shows the definition, call, and expansion of a macro.

Example 6-1. Macro Definition, Call, and Expansion

```
       1              *  add3 arg1, arg2, arg3
       2              *  arg3 = arg1 + arg2 + arg3
       3
       4              add3    .macro P1, P2, P3, ADDRP
       5
       6                      MOV   ACC, P1
       7                      ADD   ACC, P2
       8                      ADD   ACC, P3
       9                      ADD   ACC, ADDRP
      10                      .endm
      11
      12                      .global ABC, def, ghi, adr
      13
      14 000000              add3  @abc, @def, @ghi, @adr
1
1        000000 E000!         MOV   ACC, @abc
1        000001 A000!         ADD   ACC, @def
1        000002 A000!         ADD   ACC, @ghi
1        000003 A000!         ADD   ACC, @adr
      15
```

6.3 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 4.8.8).

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 6.3.6.

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 6-2 shows the expansion of a macro with varying numbers of arguments.

Example 6-2. Calling a Macro With Varying Numbers of Arguments

Macro definition:

```
Parms    .macro      a,b,c
;                a = :a:
;                b = :b:
;                c = :c:
      .endm
```

Calling the macro:

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

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

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

6.3.1 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 6-3 shows character strings being assigned to substitution symbols.

Example 6-3. The .asg Directive

```
.asg   "A4", RETVAL       ; return value
```

- 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 6-4 shows arithmetic being performed on substitution symbols.

Example 6-4. The .eval Directive

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

In Example 6-4, the .asg directive could be replaced with the .eval directive (.eval 1, counter) without changing the output. In simple cases like this, you can use .eval and .asg interchangeably. However, you must use .eval if you want to calculate a *value* from an expression. While .asg only assigns a character string to a substitution symbol, .eval evaluates an expression and then assigns the character string equivalent to a substitution symbol.

See Assign a Substitution Symbol for more information about the .asg and .eval assembler directives.

6.3.2 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 6-1, *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 6-1. 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 4.8.6.

Example 6-5 shows built-in substitution symbol functions.

Example 6-5. Using Built-In Substitution Symbol Functions

```
1       global   x, label
2               .asg   label, ADDR             ; ADDR = label
3               .if    ($symcmp(ADDR,"label") = 0) ; evaluates to true
4 000000 8000!  SUB    ACC, @ADDR
5               .endif
6               .asg   "x, y, z", list         ; list = x, y, z
7               .if    ($ismember(ADDR, list)) ; ADDR = x list = y,z
8 000001 8000!  SUB    ACC, @ADDR
9               .endif
```

6.3.3 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 6-6, the x is substituted for z; z is substituted for y; and y is substituted for x. The assembler recognizes this as infinite recursion and ceases substitution.

Example 6-6. Recursive Substitution

```
      1                 .global x
      2                 .asg  "x", z  ; declare z and assign z = "x"
      3                 .asg  "z", y  ; declare y and assign y = "z"
      4                 .asg  "y", x  ; declare x and assign x = "y"
      5 000000 FF10     ADD   ACC, x
        000001 0000!
      6
```

6.3.4 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 6-7 shows how the forced substitution operator is used.

Example 6-7. Using the Forced Substitution Operator

```
force    .macro   x
         .loop    8
PORT:x:  .set     x*4
         .eval    x+1, x
         .endloop
         .endm

         .global  portbase
         force

  PORT0  .set     0
  PORT1  .set     4
    .
    .
    .
  PORT7  .set     28
```

6.3.5 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 6-8 and Example 6-9 show built-in substitution symbol functions used with subscripted substitution symbols. In Example 6-8, subscripted substitution symbols redefine the STW instruction so that it handles immediates. In Example 6-9, the subscripted substitution symbol is used to find a substring strg1 beginning at position start in the string strg2. The position of the substring strg1 is assigned to the substitution symbol pos.

Example 6-8. Using Subscripted Substitution Symbols to Redefine an Instruction

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

ADDX        #100                      ;macro call
```

Example 6-9. Using Subscripted Substitution Symbols to Find Substrings

```
substr   .macro  start,strg1,strg2,pos
         .var    len1,len2,i,tmp
         .if     $symlen(start) = 0
         .eval   1,start
         .endif
         .eval   0,pos
         .eval   start,i
         .eval   $symlen(strg1),len1
         .eval   $symlen(strg2),len2
         .loop
         .break  i = (len2 - len1 + 1)
         .asg    ":strg2(i,len1):",tmp
         .if     $symcmp(strg1,tmp) = 0
         .eval   i,pos
         .break
         .else
         .eval   i + 1,i
         .endif
         .endloop
         .endm

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

6.3.6 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 6-8 and Example 6-9.

6.4 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 the same way it expands other macros. See Section 6.1 for how the assembler expands macros. You can control the listing of library entry expansions with the .mlist directive. For information about the .mlist directive, see Section 6.8 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 7.1.

6.5 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 5.7.

Example 6-10, Example 6-11, and Example 6-12 show the .loop/.break/ .endloop directives, properly nested conditional assembly directives, and built-in substitution symbol functions used in a conditional assembly code block.

Example 6-10. The .loop/.break/.endloop Directives

```
.asg    1,x
.loop

.break  (x == 10)  ; if x == 10, quit loop/break with expression

.eval   x+1,x
.endloop
```

Example 6-11. Nested Conditional Assembly Directives

```
      .asg    1,x
      .loop

      .if     (x == 10)  ; if x == 10, quit loop
      .break  (x == 10)  ; force break
      .endif

      .eval   x+1,x
      .endloop
```

Example 6-12. Built-In Substitution Symbol Functions in a Conditional Assembly Code Block

```
MACK3   .macro    src1, src2, sum, k

;        sum = sum + k * (src1 * src2)

        .if    k = 0
        MOV    T,#src1
        MPY    ACC,T,#src2
        MOV    DP,#sum
        ADD    @sum,AL
        .else
        MOV    T,#src1
        MPY    ACC,T,#k
        MOV    T,AL
        MPY    ACC,T,#src2
        MOV    DP,#sum
        ADD    @sum,AL
        .endif

        .endm

        .global A0, A1, A2

        MACK3  A0,A1,A2,0
        MACK3  A0,A1,A2,100
```

6.6 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. See Section 4.8.3 for more about labels.

The syntax for a unique label is:

label **?**

Example 6-13 shows 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 4.3).

Example 6-13. Unique Labels in a Macro

```
       1
       2              min     .macro  x, y, z
       3
       4                      MOV    z, y
       5                      CMP   x, y
       6                      B     l?,GT
       7                      MOV    z, x
       8              l?
       9                      .endm
      10
      11 00000000               min AH, AL, PH
1
1        00000000 2FA9           MOV    PH, AL
1        00000001 55A9           CMP   AH, AL
1        00000002 6202           B     l?,GT
1        00000003 2FA8           MOV    PH, AH
1                     l?
      12

LABEL                                VALUE       DEFN    REF

.TMS320C2800                        000001          0
.TMS320C2800_FPU32                  000000          0
__TI_ASSEMBLER_VERSION_QUAL_ID__    001c52          0
__TI_ASSEMBLER_VERSION_QUAL__       000049          0
__TI_ASSEMBLER_VERSION__            4c4f28          0
l$1$                                000004'        12     11
```

6.7 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 6-14 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 6-14. Producing Messages in a Macro

```
       1              testparam   .macro  x, y
       2              !
       3              ! This macro checks for the correct number of parameters.
       4              ! It generates an error message if x and y are not present.
       5              !
       6              ! The first line tests for proper input.
       7              !
       8                  .if     ($symlen(x) == 0)
       9                  .emsg   "ERROR --missing parameter in call to TEST"
      10                  .mexit
      11                  .else
      12                   MOV       ACC, #2
      13                   MOV       AL, #1
      14                   ADD       ACC, @AL
      15                  .endif
      16                  .endm
      17
      18 000000           testparam   1, 2
1                         .if     ($symlen(x) == 0)
1                         .emsg   "ERROR --missing parameter in call to TEST"
1                         .mexit
1                         .else
1       000000 FF20         MOV       ACC, #2
        000001 0002
1       000002 9A01         MOV       AL, #1
1       000003 A0A9         ADD       ACC, @AL
1                         .endif
```

6.8 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.

6.9 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 6-15 shows nested macros. The y in the in_block macro hides the y in the out_block macro. The x and z from the out_block macro, however, are accessible to the in_block macro.

Example 6-15. Using Nested Macros

```
in_block  .macro y,a
             .          ; visible parameters are y,a and x,z from the calling macro
          .endm

out_block .macro    x,y,z
             .          ; visible parameters are x,y,z
             .
          in_block x,y  ; macro call with x and y as arguments
             .
             .
          .endm
          out_block     ; macro call
```

Example 6-16 shows recursive and fact macros. The fact macro produces assembly code necessary to calculate the factorial of n, where n is an immediate value. The result is placed in the A register. The fact macro accomplishes this by calling fact1, which calls itself recursively.

Example 6-16. Using Recursive Macros

```
 1                      .fcnolist
 2
 3              fact    .macro N, LOC
 4
 5                      .if N < 2
 6                      MOV    @LOC, #1
 7                      .else
 8                      MOV    @LOC, #N
 9
10
11                      .eval N-1, N
12                 fact1
13
14                      .endif
15                      .endm
16
17              fact1   .macro
18                      .if N > 1
19                      MOV    @T, @LOC
20                      MPYB   @P, @T, #N
21                      MOV    @LOC, @P
22                      MOV    ACC, @LOC
23                      .eval  N - 1, N
24                 fact1
25
26                      .endif
27                      .endm
```

6.10 Macro Directives Summary

The directives listed in Table 6-2 through Table 6-6 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 6-2. Creating Macros

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.endm	End macro definition	Section 6.2	.endm
macname **.macro** [*parameter*$_1$][,... , *parameter*$_n$]	Define macro by *macname*	Section 6.2	.macro
.mexit	Go to .endm	Section 6.2	Section 6.2
.mlib *filename*	Identify library containing macro definitions	Section 6.4	.mlib

Table 6-3. Manipulating Substitution Symbols

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.asg ["]*character string*["], *substitution symbol*	Assign character string to substitution symbol	Section 6.3.1	.asg
.eval *well-defined expression, substitution symbol*	Perform arithmetic on numeric substitution symbols	Section 6.3.1	.eval
.var *sym*$_1$ [, *sym*$_2$, ..., *sym*$_n$]	Define local macro symbols	Section 6.3.6	.var

Table 6-4. Conditional Assembly

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.break [*well-defined expression*]	Optional repeatable block assembly	Section 6.5	.break
.endif	End conditional assembly	Section 6.5	.endif
.endloop	End repeatable block assembly	Section 6.5	.endloop
.else	Optional conditional assembly block	Section 6.5	.else
.elseif *well-defined expression*	Optional conditional assembly block	Section 6.5	.elseif
.if *well-defined expression*	Begin conditional assembly	Section 6.5	.if
.loop [*well-defined expression*]	Begin repeatable block assembly	Section 6.5	.loop

Table 6-5. Producing Assembly-Time Messages

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.emsg	Send error message to standard output	Section 6.7	.emsg
.mmsg	Send assembly-time message to standard output	Section 6.7	.mmsg
.wmsg	Send warning message to standard output	Section 6.7	.wmsg

Table 6-6. Formatting the Listing

Mnemonic and Syntax	Description	See	
		Macro Use	Directive
.fclist	Allow false conditional code block listing (default)	Section 6.8	.fclist
.fcnolist	Suppress false conditional code block listing	Section 6.8	.fcnolist
.mlist	Allow macro listings (default)	Section 6.8	.mlist
.mnolist	Suppress macro listings	Section 6.8	.mnolist
.sslist	Allow expanded substitution symbol listing	Section 6.8	.sslist
.ssnolist	Suppress expanded substitution symbol listing (default)	Section 6.8	.ssnolist

Archiver Description

The TMS320C28x 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 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**

7.1 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.

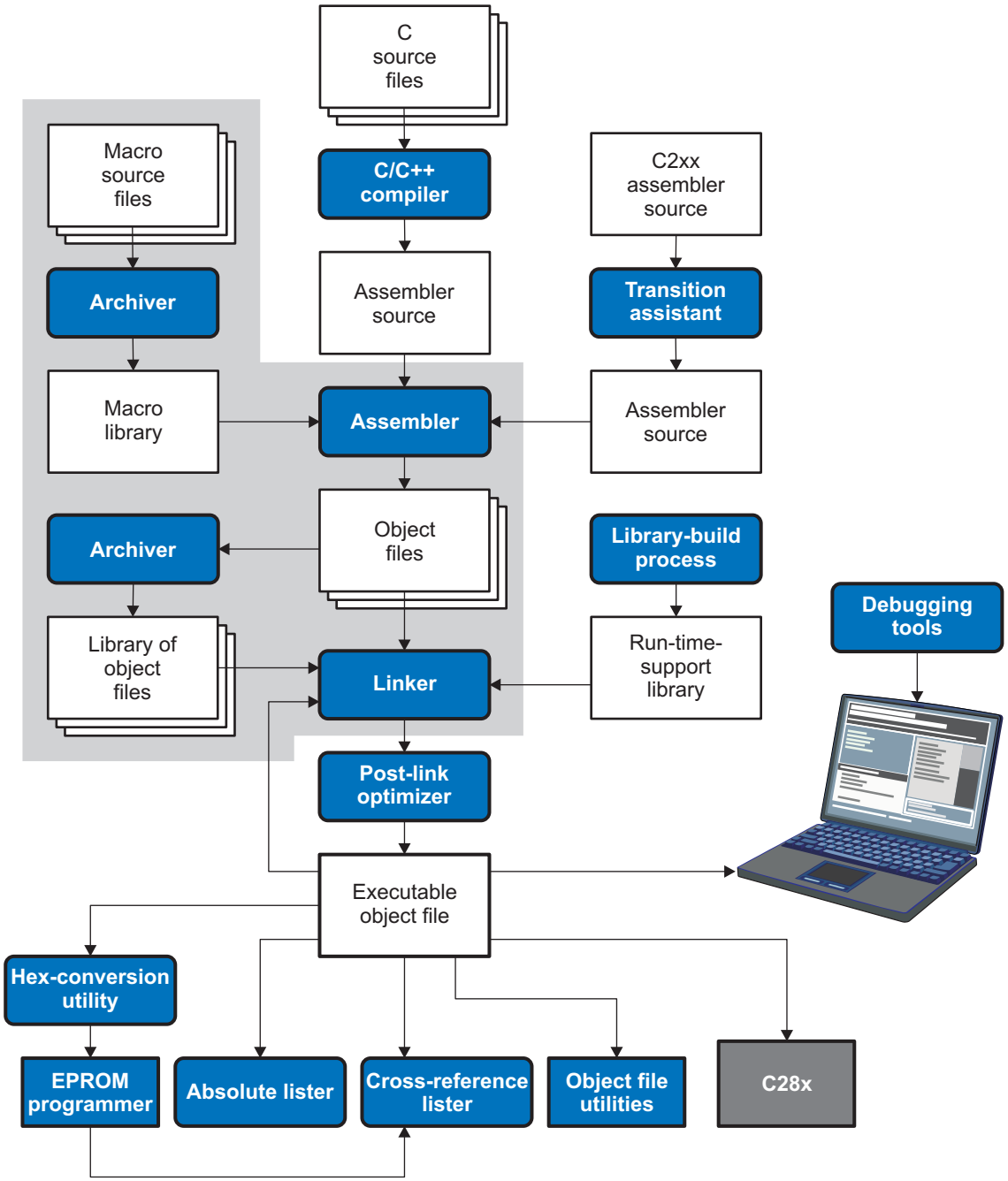
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The 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 6 discusses macros and macro libraries in detail, while this chapter explains how to use the archiver to build libraries.

7.2 The Archiver's Role in the Software Development Flow

Figure 7-1 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 7-1. The Archiver in the TMS320C28x Software Development Flow



7.3 Invoking the Archiver

To invoke the archiver, enter:

ar2000[-]*command* [*options*] *libname* [*filename*$_1$ ... *filename*$_n$]

ar2000 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 7-1 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:

- **-h** provide command-line help
- **-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.

7.4 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:

  ```
  ar2000 -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:

  ```
  ar2000 -t function
  ```

  The archiver responds as follows:

  ```
     SIZE   DATE                     FILE NAME
  --------   ------------------------ ----------------
     4260   Thu Mar 28 15:38:18 2019 sine.obj
     4260   Thu Mar 28 15:38:18 2019 cos.obj
     4260   Thu Mar 28 15:38:18 2019 flt.obj
  ```

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

  ```
  ar2000 -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.

  ```
  ar2000 -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:

  ```
  ar2000 -r macros push.asm
  ```

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

  ```
  ar2000 -@modules.cmd
  ```

  The archiver responds as follows:

  ```
  ==>  building archive 'modules.lib'
  ```

Example 7-1 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 7-1. 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
```

7.5 Library Information Archiver Description

Section 7.1 through Section 7.4 explain 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. However, if you have several versions of a library, it can be 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.

7.5.1 Invoking the Library Information Archiver

To invoke the library information archiver, enter:

libinfo2000 [*options*] **--output**=*libname libname*$_1$ [*libname*$_2$ ... *libname*$_n$]

libinfo2000 is the command that invokes the library information archiver.

options changes the default behavior of the library information archiver. These options are:

- **--output** *libname* specifies the name of the index library to create or update. This option is required.
- **--update** updates any existing information in the index library specified with the --output 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.

7.5.2 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
mylib_2800_ml.lib	(default)
mylib_2800_fpu32.lib	--float_support=fpu32

Using the library information archiver, you can create an index library called mylib.lib from the above libraries:

```
libinfo2000 --output mylib.lib mylib_2800.lib mylib_2800_fpu32.lib mylib_2800_ml.lib
```

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.

- **Example 1**:

```
cl2000 --issue_remarks main.c -z -l lnk.cmd ./mylib.lib
<Linking>
remark: linking in "mylib_2800.lib" in place of "mylib.lib"
```

- **Example 2** (with FPU32 support):

```
cl2000 --float_support=fpu32 --issue_remarks main.c -z -l lnk.cmd ./mylib.lib
<Linking>
remark: linking in "mylib_2800_fpu32.lib" in place of "mylib.lib"
```

7.5.3 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.

7.5.4 Requirements

You must follow these requirements to use library index files:

- At least one application object file 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 TMS320C28x 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 includes a detailed discussion of sections.

Topic **Page**

8.1 Linker Overview

The TMS320C28x linker allows you to allocate output sections efficiently in 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

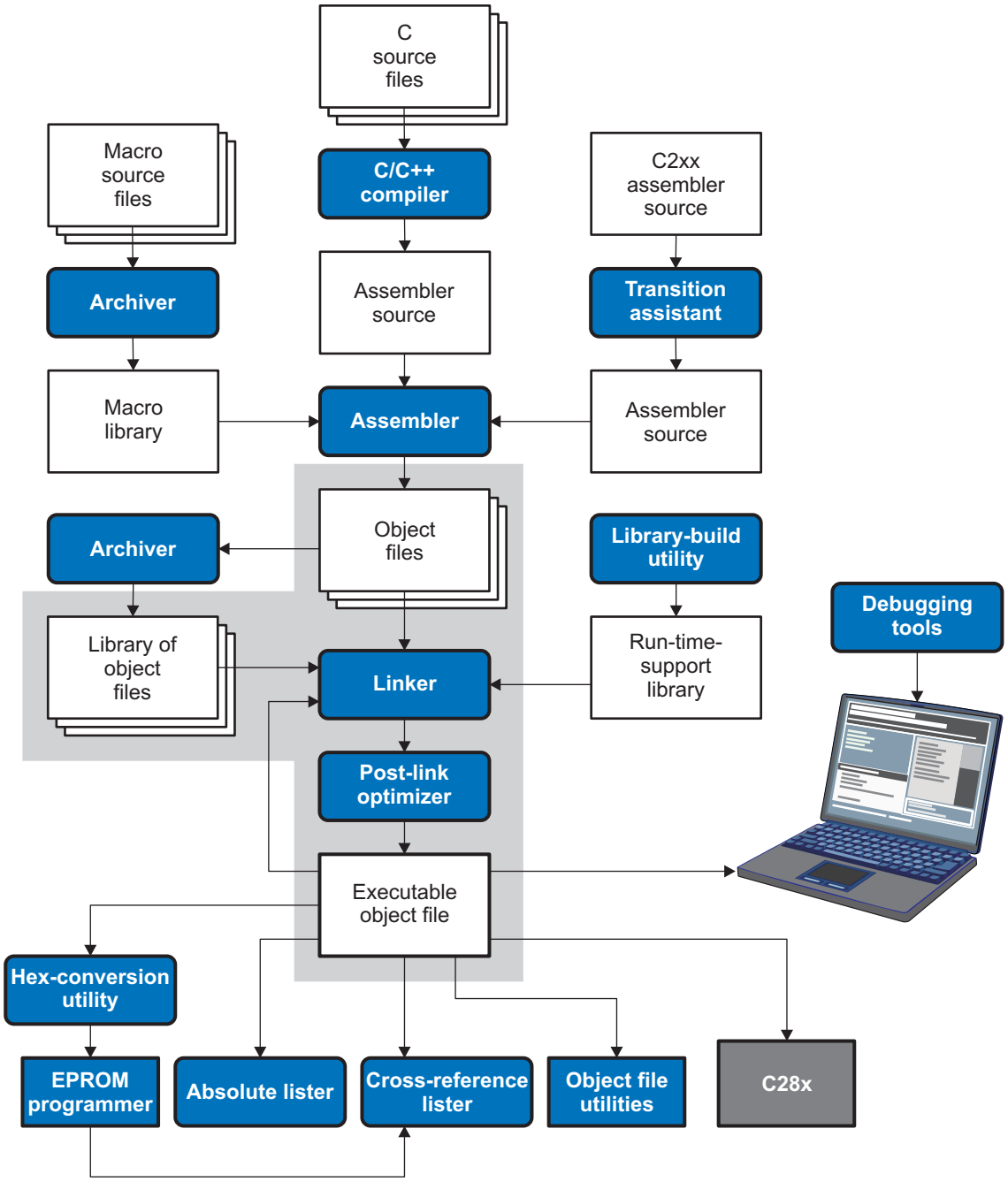
The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow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

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

8.2 The Linker's Role in the Software Development Flow

Figure 8-1 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 TMS320C28x device.

Figure 8-1. The Linker in the TMS320C28x Software Development Flow



8.3 Invoking the Linker

The general syntax for invoking the linker is:

cl2000 --run_linker [*options*] *filename*$_1$ .... *filename*$_n$

cl2000 --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 linker command file. (Options are discussed in Section 8.4.)
filename $_1$, *filename* $_n$	can be object files, linker command files, or archive libraries. The default extensions for input files are .c.obj (for C source files) and .cpp.obj (for C++ source files). 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.

NOTE: The default file extensions for object files created by the compiler have been changed. Object files generated from C source files have the .c.obj extension. Object files generated from C++ source files have the .cpp.obj extension. Object files generated from assembly source files still have the .obj extension.

There are two methods for invoking the linker:

- Specify options and filenames on the command line. This example links two files, file1.c.obj and file2.c.obj, and creates an output module named link.out.

  ```
  cl2000 --run_linker file1.c.obj file2.c.obj --output_file=link.out
  ```

- Put filenames and options in a linker command file. Filenames that are specified inside a linker command file must begin with a letter. For example, assume the file linker.cmd contains the following lines:

  ```
  --output_file=link.out file1.c.obj file2.c.obj
  ```

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

  ```
  cl2000 --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:

  ```
  cl2000 --run_linker --map_file=link.map linker.cmd file3.c.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.c.obj, file2.c.obj, and file3.c.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 8.11.

8.4 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 8-1. Basic Options Summary

Option	Alias	Description	Section
--run_linker	-z	Enables linking	Section 8.3
--output_file	-o	Names the executable output module. The default filename is a.out.	Section 8.4.24
--map_file	-m	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename*	Section 8.4.19
--stack_size	-stack	Sets C system stack size to *size* words and defines a global symbol that specifies the stack size. Default = 1K words	Section 8.4.30
--heap_size	-heap	Sets heap size (for the dynamic memory allocation in C) to *size* words and defines a global symbol that specifies the heap size. Default = 1K words	Section 8.4.15

Table 8-2. File Search Path Options Summary

Option	Alias	Description	Section
--library	-l	Names an archive library or link command *filename* as linker input	Section 8.4.17
--disable_auto_rts		Disables the automatic selection of a run-time-support library	Section 8.4.8
--priority	-priority	Satisfies unresolved references by the first library that contains a definition for that symbol	Section 8.4.17.3
--reread_libs	-x	Forces rereading of libraries, which resolves back references	Section 8.4.17.3
--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 8.4.17.1

Table 8-3. Command File Preprocessing Options Summary

Option	Alias	Description	Section
--define		Predefines *name* as a preprocessor macro.	Section 8.4.11
--undefine		Removes the preprocessor macro *name*.	Section 8.4.11
--disable_pp		Disables preprocessing for command files	Section 8.4.11

Table 8-4. Diagnostic Options Summary

Option	Alias	Description	Section
--diag_error		Categorizes the diagnostic identified by *num* as an error	Section 8.4.7
--diag_remark		Categorizes the diagnostic identified by *num* as a remark	Section 8.4.7
--diag_suppress		Suppresses the diagnostic identified by *num*	Section 8.4.7
--diag_warning		Categorizes the diagnostic identified by *num* as a warning	Section 8.4.7
--display_error_number		Displays a diagnostic's identifiers along with its text	Section 8.4.7
--emit_references:file[=*file*]		Emits a file containing section information. The information includes section size, symbols defined, and references to symbols.	Section 8.4.7
--emit_warnings_as_errors	-pdew	Treats warnings as errors	Section 8.4.7
--issue_remarks		Issues remarks (nonserious warnings)	Section 8.4.7
--no_demangle		Disables demangling of symbol names in diagnostics	Section 8.4.21
--no_warnings		Suppresses warning diagnostics (errors are still issued)	Section 8.4.7
--set_error_limit		Sets the error limit to *num*. The linker abandons linking after this number of errors. (The default is 100.)	Section 8.4.7
--verbose_diagnostics		Provides verbose diagnostics that display the original source with line-wrap	Section 8.4.7
--warn_sections	-w	Displays a message when an undefined output section is created	Section 8.4.34

Table 8-5. 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 8.4.3.1
--ecc={ on | off }		Enable linker-generated Error Correcting Codes (ECC). The default is off.	Section 8.4.12 Section 8.5.10
--ecc:data_error		Inject the specified errors into the output file for testing	Section 8.4.12 Section 8.5.10
--ecc:ecc_error		Inject the specified errors into the Error Correcting Code (ECC) for testing	Section 8.4.12 Section 8.5.10
--mapfile_contents		Controls the information that appears in the map file.	Section 8.4.20
--relocatable	-r	Produces a nonexecutable, relocatable output module	Section 8.4.3.2
--rom	-r	Create a ROM object	
--run_abs	-abs	Produces an absolute listing file	Section 8.4.28
--xml_link_info		Generates a well-formed XML *file* containing detailed information about the result of a link	Section 8.4.35

Table 8-6. 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 8.4.13
--globalize		Changes the symbol linkage to global for symbols that match *pattern*	Section 8.4.18
--hide		Hides global symbols that match *pattern*	Section 8.4.16
--localize		Changes the symbol linkage to local for symbols that match *pattern*	Section 8.4.18
--make_global	-g	Makes *symbol* global (overrides -h)	Section 8.4.18.1
--make_static	-h	Makes all global symbols static	Section 8.4.18.1
--no_sym_merge	-b	Disables merge of symbolic debugging information in COFF object files	Section 8.4.22
--no_symtable	-s	Strips symbol table information and line number entries from the output module	Section 8.4.23
--retain		Retains a list of sections that otherwise would be discarded. (EABI only)	Section 8.4.27
--scan_libraries	-scanlibs	Scans all libraries for duplicate symbol definitions	Section 8.4.29
--symbol_map		Maps symbol references to a symbol definition of a different name	Section 8.4.32
--undef_sym	-u	Places an unresolved external *symbol* into the output module's symbol table	Section 8.4.33
--unhide		Reveals (un-hides) global symbols that match *pattern*	Section 8.4.16

Table 8-7. Run-Time Environment Options Summary

Option	Alias	Description	Section
--arg_size	--args	Allocates memory to be used by the loader to pass arguments	Section 8.4.4
--fill_value	-f	Sets default fill values for holes within output sections; *fill_value* is a 32-bit constant	Section 8.4.14
--ram_model	-cr	Initializes variables at load time	Section 8.4.26
--rom_model	-c	Autoinitializes variables at run time	Section 8.4.26

Table 8-8. Link-Time Optimization Options Summary

Option	Alias	Description	Section
--cinit_compression [=compression_kind]		Specifies the type of compression to apply to the C auto initialization data. The default *compression_kind* if this option is used with no kind specified is lzss for Lempel-Ziv-Storer-Szymanski compression. Alternately, specify --cinit_compression=rle to use Run Length Encoded compression, which generally provides less efficient compression. (EABI only)	Section 8.4.5
--compress_dwarf		Aggressively reduces the size of DWARF information from input object files. (EABI only)	Section 8.4.6
--copy_compression [=compression_kind]		Compresses data copied by linker copy tables (EABI only)	Section 8.4.5
--unused_section_elimination		Eliminates sections that are not needed in the executable module; on by default. (EABI only)	Section 8.4.10
--keep_asm		Retain any post-link files (.pl) and .absolute listing files (.abs) generated by the −plink option. This allows you to view any changes the post-link optimizer makes. (Requires use of -plink)	Note (1)
--no_postlink_across_calls	-nf	Disable post-link optimizations across functions. (Requires use of -plink)	Note (1)
--plink_advice_only		Annotates assembly code with comments if changes cannot be made safely due to pipeline considerations, such as when float support or VCU support is enabled. (Requires use of -plink)	Note (1)
--postlink_exclude	-ex	Exclude files from post-link pass. (Requires use of -plink)	Note (1)
--postlink_opt	-plink	Post-link optimizations. (Only after --run_linker or -z)	Note (1)

(1) For more information, refer to the *Post-Link Optimizer* chapter in the *TMS320C28x Optimizing C /C++ Compiler v6.0 User's Guide*.

Table 8-9. Miscellaneous Options Summary

Option	Alias	Description	Section
--disable_clink	-j	Disables conditional linking of COFF object modules. (COFF only)	Section 8.4.9
--linker_help	-help	Displays information about syntax and available options	–
--preferred_order		Prioritizes placement of functions	Section 8.4.25
--strict_compatibility		Performs more conservative and rigorous compatibility checking of input object files	Section 8.4.31
--zero_init		Controls preinitialization of uninitialized variables. Default is on. (EABI only)	Section 8.4.36

8.4.1 Wildcards in File, Section, and Symbol Patterns

The linker allows file, section, and symbol names to be specified using the asterisk (*) and question mark (?) wildcards. Using * matches any number of characters and using ? matches a single character. Using wildcards 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.c.obj, taskX.o55, etc. */

SECTIONS
{
   .fast_code: { *.obj(*fast*) }              > FAST_MEM
   .vectors  : { vectors.c.obj(.vector:part1:*) > 0xFFFFFF00
   .str_code : { rts*.lib<str*.c.obj>(.text) }  > S1ROM
}
```

8.4.2 Specifying C/C++ Symbols with Linker Options

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. For example, the --localize and --globalize options accept link-time symbols. Thus, you specify --localize='_foo2' to localize the C function _foo2().

For EABI, the link-time symbol is the same as the C/C++ identifier name. The compiler *does not* prepend an underscore to the beginning of C/C++ identifiers.

For more information on referencing symbol names, see the "Object File Symbol Naming Conventions (Linknames)" section in the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

For information specifically about C++ symbol naming, see Section 13.3.1 in this document and the "C++ Name Demangler" chapter in the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

See Section 8.6.1 for information about referring to linker symbols in C/C++ code.

8.4.3 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 (Section 2.7).

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).

8.4.3.1 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 8.5.11.4)
- An header that describes information such as the program entry point (optional in COFF)
- *No* unresolved references

The following example links file1.c.obj and file2.c.obj and creates an absolute output module called a.out:

```
cl2000 --run_linker --absolute_exe file1.c.obj file2.c.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.

8.4.3.2 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.c.obj and file2.c.obj and creates a relocatable output module called a.out:

```
cl2000 --run_linker --relocatable file1.c.obj file2.c.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 8.10.)

8.4.3.3 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.c.obj and file2.c.obj to create an executable, relocatable output module called xr.out:

```
cl2000 --run_linker -ar file1.c.obj file2.c.obj --output_file=xr.out
```

8.4.4 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 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 *TMS320C28x Optimizing C/C++ Compiler User's Guide* for information about the loader.

8.4.5 Compression (--cinit_compression and --copy_compression Option)

By default, the linker does not compress copy table (Section 3.3.3 and Section 8.8) source data sections. The --cinit_compression and --copy_compression options specify compression through the linker. These options are supported only when you use EABI by specifying the --abi=eabi option. They are not supported for COFF object modules.

The --cinit_compression option specifies the compression type the linker applies to the C autoinitialization copy table source data sections. The default is lzss.

Overlays can be managed by using linker-generated copy tables. To save ROM space the linker can compress the data copied by the copy tables. The compressed data is decompressed during copy. The --copy_compression option controls the compression of the copy data tables.

The syntax for the options are:

--cinit_compression[=*compression_kind*]

--copy_compression[=*compression_kind*]

The *compression_kind* can be one of the following types:

- **off**. Don't compress the data.
- **rle**. Compress data using Run Length Encoding.
- **lzss**. Compress data using Lempel-Ziv-Storer-Szymanski compression (the default if no *compression_kind* is specified).

See Section 8.8.5 for more information about compression.

8.4.6 Compress DWARF Information (--compress_dwarf Option)

The --compress_dwarf option aggressively reduces the size of DWARF information by eliminating duplicate information from input object files.

For COFF files, this is the default behavior, and can be disabled for COFF with the legacy --no_sym_merge option.

With EABI, the default is to use DWARF 4, which merges types efficiently. Such merging can be made slightly more efficient using the --compress_dwarf option. (See the ELF specification for information on COMDAT groups and type merging.)

8.4.7 Control Linker Diagnostics

The linker honors certain C/C++ compiler options to control linker-generated diagnostics. The diagnostic options must be specified before the --run_linker option.

--diag_error=*num* Categorize the diagnostic identified by *num* as an error. To find 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* Categorize the diagnostic identified by *num* as a remark. To find 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* Suppress the diagnostic identified by *num*. To find 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* Categorize the diagnostic identified by *num* as a warning. To find 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	Display 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 *TMS320C28x Optimizing C/C++ Compiler User's Guide* for more information on understanding diagnostic messages.
--emit_references:file [=*filename*]	Emits a file containing section information. The information includes section size, symbols defined, and references to symbols. This information allows you to determine why each section is included in the linked application. The output file is a simple ASCII text file. The *filename* is used as the base name of a file created. For example, --emit_references:file=myfile generates a file named myfile.txt in the current directory.
--emit_warnings_as_errors	Treat 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	Issue remarks (nonserious warnings), which are suppressed by default.
--no_warnings	Suppress warning diagnostics (errors are still issued).
--set_error_limit=*num*	Set 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	Provide verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line

8.4.8 Automatic Library Selection (--disable_auto_rts Option)

The --disable_auto_rts option disables the automatic selection of a run-time-support (RTS) library. See the *TMS320C28x Optimizing C/C++ Compiler User's Guide* for details on the automatic selection process.

8.4.9 Disable Conditional Linking (--disable_clink Option)

NOTE: This option is supported for COFF mode only.

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.

The --disable_clink option is not used with EABI.

8.4.10 Do Not Remove Unused Sections (--unused_section_elimination Option)

NOTE: This option applies when using EABI mode only. It is ignored when using COFF mode.

In order to minimize the foot print, the ELF linker does not include a section that is not needed to resolve any references in the final executable. Use --unused_section_elimination=off to disable this optimization. The syntax for the option is:

--unused_section_elimination[=*on*|*off*]

The linker default behavior is equivalent to --unused_section_elimination=on.

8.4.11 Linker Command File Preprocessing (--disable_pp, --define and --undefine Options)

The linker preprocesses linker 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:

```
cl2000 --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 8.4.17)

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 provides the built-in macro definitions listed in Table 8-10. The availability of these macros within the linker is determined by the command-line options used, not the build attributes of the files being linked. If these macros are not set as expected, confirm that your project's command line uses the correct compiler option settings.

Table 8-10. Predefined C28x Macro Names

Macro Name	Description
_ _DATE_ _	Expands to the compilation date in the form *mmm dd yyyy*
_ _FILE_ _	Expands to the current source filename
_ _TI_COMPILER_VERSION_ _	Defined to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.
_ _TI_EABI_ _	Defined to 1 if EABI is enabled; otherwise, it is undefined.
_ _TIME_ _	Expands to the compilation time in the form "*hh:mm:ss*"
_ _TMS320C2000_ _	Defined for all C2000 processors
_ _TMS320C28XX_ _	Defined if target is C28x
_ _TMS320C28XX_CLA_ _	Defined to 1 if and --cla_support is used; otherwise it is undefined.
_ _TMS320C28XX_CLA0_ _	Defined to 1 if --cla_support=cla0 is used; otherwise it is undefined.
_ _TMS320C28XX_CLA1_ _	Defined to 1 if --cla_support=cla1 is used; otherwise it is undefined.
_ _TMS320C28XX_CLA2_ _	Defined to 1 if --cla_support=cla2 is used; otherwise it is undefined.
_ _TMS320C28XX_FPU32_ _	Defined to 1 if --float_support=fpu32 is used; otherwise it is undefined.
_ _TMS320C28XX_FPU64_ _	Defined to 1 if --float_support=fpu64 is used; otherwise it is undefined.
_ _TMS320C28XX_IDIV_ _	Defined to 1 if --idiv_support=idiv0 is used; otherwise it is undefined.
_ _TMS320C28XX_TMU_ _	Defined to 1 if --tmu_support is used with any setting; otherwise it is undefined.
_ _TMS320C28XX_TMU0_ _	Defined to 1 if --tmu_support is used with any setting; otherwise it is undefined.
_ _TMS320C28XX_TMU1_ _	Defined to 1 if --tmu_support=tmu1 is used; otherwise it is undefined.
_ _TMS320C28XX_VCU0_ _	Defined to 1 if --vcu_support=vcu0 is used; otherwise it is undefined.
_ _TMS320C28XX_VCU2_ _	Defined to 1 if --vcu_support=vcu2 is used; otherwise it is undefined.
_ _TMS320C28XX_VCRC_ _	Defined to 1 if --vcu_support=vcrc is used; otherwise it is undefined.

8.4.12 Error Correcting Code Testing (--ecc Options)

Error Correcting Codes (ECC) can be generated and placed in separate sections through the linker command file.

To enable ECC support, include **--ecc=on** as a linker option on the command line. By default ECC generation is off, even if the ECC directive and ECC specifiers are used in the linker command file. This allows you to fully configure ECC in the linker command file while still being able to quickly turn the code generation on and off via the command line. See Section 8.5.10 for details on linker command file syntax to configure ECC support.

ECC uses extra bits to allow errors to be detected and/or corrected by a device. The ECC support provided by the linker is compatible with the ECC support in TI Flash memory on various TI devices. TI Flash memory uses a modified Hamming(72,64) code, which uses 8 parity bits for every 64 bits. Check the documentation for your Flash memory to see if ECC is supported. (ECC for read-write memory is handled completely in hardware at run time.)

After enabling ECC with the --ecc=on option, you can use the following command-line options to test ECC by injecting bit errors into the linked executable. These options let you specify an address where an error should appear and a bitmask of bits in the code/data at that address to flip. You can specify the address of the error absolutely or as an offset from a symbol. When a data error is injected, the ECC parity bits for the data are calculated as if the error were not present. This simulates bit errors that might actually occur and tests ECC's ability to correct different levels of errors.

The **--ecc:data_error option** injects errors into the load image at the specified location. The syntax is:

```
--ecc:data_error=(symbol+offset|address)[,page],bitmask
```

The *address* is the location of the minimum addressable unit where the error is to be injected. A *symbol+offset* can be used to specify the location of the error to be injected with a signed offset from that symbol. The *page* number is needed to make the location non-ambiguous if the address occurs on multiple memory pages. The *bitmask* is a mask of the bits to flip; its width should be the width of an addressable unit.

For example, the following command line flips the least-significant bit in the byte at the address 0x100, making it inconsistent with the ECC parity bits for that byte:

```
cl2000 test.c --ecc:data_error=0x100,0x01 -z -o test.out
```

The following command flips two bits in the third byte of the code for main():

```
cl2000 test.c --ecc:data_error=main+2,0x42 -z -o test.out
```

The **--ecc:ecc_error option** injects errors into the ECC parity bits that correspond to the specified location. Note that the ecc_error option can therefore only specify locations inside ECC input ranges, whereas the data_error option can also specify errors in the ECC output memory ranges. The syntax is:

```
--ecc:ecc_error=(symbol+offset|address)[,page],bitmask
```

The parameters for this option are the same as for --ecc:data_error, except that the *bitmask* must be exactly 8 bits. Mirrored copies of the affected ECC byte will also contain the same injected error.

An error injected into an ECC byte with --ecc:ecc_error may cause errors to be detected at run time in any of the 8 data bytes covered by that ECC byte.

For example, the following command flips every bit in the ECC byte that contains the parity information for the byte at 0x200:

```
cl2000 test.c --ecc:ecc_error=0x200,0xff -z -o test.out
```

The linker disallows injecting errors into memory ranges that are neither an ECC range nor the input range for an ECC range. The compiler can only inject errors into initialized sections.

8.4.13 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 *TMS320C28x 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.c.obj and file2.c.obj. The symbol begin is the entry point; begin must be defined as external in file1 or file2.

```
cl2000 --run_linker --entry_point=begin file1.c.obj file2.c.obj
```

See Section 8.6.1 for information about referring to linker symbols in C/C++ code.

8.4.14 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:

```
cl2000 --run_linker --fill_value=0xABCDABCD file1.c.obj file2.c.obj
```

8.4.15 Define Heap Size (--heap_size Option)

The C/C++ compiler uses an uninitialized section called .esysmem 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 word heap:

```
cl2000 --run_linker --heap_size=0x1000 /* defines a 4k heap (.esysmem section)*/
```

The linker creates the .esysmem section only if there is a .esysmem section in an input file.

The linker also creates a global symbol, __SYSMEM_SIZE (for COFF) or __TI_SYSMEM_SIZE (for EABI), and assigns it a value equal to the size of the heap. The default size is 1K words. See Section 8.6.1 for information about referring to linker symbols in C/C++ code. For more about C/C++ linking, see Section 8.11.

8.4.16 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 (see Section 8.4.18), 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 "glob" (a string with optional ? or * wildcards). Use ? to match a single character. Use * to match zero or more characters.

The --hide option hides global symbols with a linkname matching the *pattern*. It hides symbols matching the pattern by changing the name to an empty string. A global symbol that 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 one does not supersede the other. Pattern A supersedes pattern B if A can match everything B can and 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.

8.4.17 Alter the Library Search Algorithm (--library Option, --search_path Option, and C2000_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. If this library defines symbols that are referenced in the file file1.c.obj, this is how you link the files:

```
cl2000 --run_linker file1.c.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 linker 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 8.5.5.5.

You can augment the linker's directory search algorithm by using the --search_path linker option or the C2000_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 C2000_C_DIR.
3. If C2000_C_DIR is not set, it searches directories named with the assembler's C2000_A_DIR environment variable.
4. It searches the current directory.

8.4.17.1 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)	`cl2000 --run_linker f1.c.obj f2.c.obj --search_path=/ld --search_path=/ld2 --library=r.lib --library=lib2.lib`
Windows	`cl2000 --run_linker f1.c.obj f2.c.obj --search_path=\ld --search_path=\ld2 --library=r.lib --library=lib2.lib`

8.4.17.2 Name an Alternate Library Directory (C2000_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 C2000_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)	**C2000_C_DIR="** *pathname*$_1$**;** *pathname*$_2$**; . . . ";** **export C2000_C_DIR**
Windows	**set C2000_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 linker command file to search for. The pathnames must follow these constraints:

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

  ```
  set C2000_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 C2000_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)	`C2000_C_DIR="/ld ;/ld2"; export C2000_C_DIR;` `cl2000 --run_linker f1.c.obj f2.c.obj --library=r.lib --library=lib2.lib`
Windows	`C2000_C_DIR=\ld;\ld2` `cl2000 --run linker f1.c.obj f2.c.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 C2000_C_DIR`
Windows	`set C2000_C_DIR=`

The assembler uses an environment variable named C2000_A_DIR to name alternate directories that contain copy/include files or macro libraries. If C2000_C_DIR is not set, the linker searches for object libraries in the directories named with C2000_A_DIR. For information about C2000_A_DIR, see Section 4.5.2. For more information about object libraries, see Section 8.6.3.

8.4.17.3 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:

```
cl2000 --run_linker --library=a.lib --library=b.lib --library=a.lib
```

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

```
cl2000 --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

% cl2000 --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 rts2800_ml.lib without providing a full replacement for rts2800_ml.lib. Using --priority and linking your new library before rts2800_ml.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.

8.4.18 Change Symbol Localization

Symbol localization changes symbol linkage from global to local (static). This is used to obscure global symbols that should not be widely visible, 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 "glob" (a string with optional ? or * wildcards). Use ? to match a single character. 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.

See Section 8.4.2 for information about using C/C++ identifiers in linker options such as --localize and --globalize.

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.

8.4.18.1 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.c.obj and file2.c.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.c.obj is treated separately from the symbol EXT defined in file2.c.obj.

```
cl2000 --run_linker --make_static file1.c.obj file2.c.obj
```

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*

8.4.19 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 linker command file:
  - **Name.** This is the name of the memory range specified with the MEMORY directive.
  - **Origin.** This specifies the starting address of a memory range.
  - **Length.** This specifies the length of a memory range.
  - **Unused.** This specifies the total amount of unused (available) memory in that memory area.
  - **Attributes.** This specifies one to four attributes associated with the named range:

    R specifies that the memory can be read.
    W specifies that the memory can be written to.
    X specifies that the memory can contain executable code.
    I specifies that the memory can be initialized.

  For more information about the MEMORY directive, see Section 8.5.4.
- A table showing the linked addresses of each output section and the input sections that make up the output sections (section placement map). This table has the following columns; this information is generated on the basis of the information in the SECTIONS directive in the linker command file:
  - **Output section.** This is the name of the output section specified with the SECTIONS directive.
  - **Origin.** The first origin listed for each output section is the starting address of that output section. The indented origin value is the starting address of that portion of the output section.
  - **Length.** The first length listed for each output section is the length of that output section. The indented length value is the length of that portion of the output section.
  - **Attributes/input sections.** This lists the input file or value associated with an output section. If the input section could not be allocated, the map file will indicate this with "FAILED TO ALLOCATE".

  For more information about the SECTIONS directive, see Section 8.5.5.
- 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.c.obj and file2.c.obj and creates a map file called map.out:

```
cl2000 --run_linker file1.c.obj file2.c.obj --map_file=map.out
```

Example 8-33 shows an example of a map file.

8.4.20 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 --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
crctables	CRC tables	On
copytables	Copy tables	On
entry	Entry point	On
load_addr	Display load addresses	Off
memory	Memory ranges	On
modules	Module view	On
sections	Sections	On
sym_defs	Defined symbols per file	Off
sym_dp	Symbols sorted by data page	On
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.

The load_addr and sym_defs attributes 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).

You can use the sym_defs filter to include information sorted on a file by file basis. You may find it useful to replace the sym_name, sym_dp, 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_dp,nosym_runaddr,sym_defs
```

By default, information about global symbols defined in an application are included in tables sorted by name, data page, and run address. If you use the --mapfile_contents=sym_defs option, static variables are also listed.

8.4.21 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.cpp.obj
```

The --no_demangle option instead shows the linkname for symbols in diagnostics. For example:

```
undefined symbol                first referenced in file
_ZN9ANewClass8getValueEv        test.cpp.obj
```

For information on referencing symbol names, see the "Object File Symbol Naming Conventions (Linknames)" section in the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

For information specifically about C++ symbol naming, see the "C++ Name Demangler" chapter in the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

8.4.22 Disable Merging 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.c.obj and f2.c.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.

8.4.23 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.c.obj and file2.c.obj and creates an output module, stripped of line numbers and symbol table information, named nosym.out:

```
cl2000 --run_linker --output_file=nosym.out --no_symtable file1.c.obj file2.c.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 strip2000 utility as described in Section 11.4.

8.4.24 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.c.obj and file2.c.obj and creates an output module named run.out:

```
cl2000 --run_linker --output_file=run.out file1.c.obj file2.c.obj
```

8.4.25 Prioritizing Function Placement (--preferred_order Option)

The compiler prioritizes the placement of a function relative to others based on the order in which --preferred_order options are encountered during the linker invocation. The syntax is:

--preferred_order=*function specification*

Refer to the *TMS320C28x Optimizing C/C++ Compiler User's Guide* for details on the program cache layout tool, which is impacted by --preferred_option.

8.4.26 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. Both options inform the linker that the program is a C program and requires a boot routine.

- 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 8.11, Section 3.3.2.1, and Section 3.3.2.2.

8.4.27 Retain Discarded Sections (--retain Option)

NOTE: The --retain option is used only with EABI mode. It is ignored when linking in COFF mode.

When --unused_section_elimination is on, the ELF linker does not include a section in the final link if it is not needed in the executable to resolve references. The --retain option tells the linker to retain a list of sections that would otherwise not be retained. This option accepts the wildcards '*' and '?'. When wildcards are used, the argument should be in quotes. The syntax for this option is:

--retain=*sym_or_scn_spec*

The --retain option take one of the following forms:

- **--retain=** *symbol_spec*

  Specifying the symbol format retains sections that define *symbol_spec*. For example, this code retains sections that define symbols that start with init:

  ```
  --retain='init*'
  ```

  You cannot specify --retain='*'.

- **--retain=** *file_spec*(*scn_spec*[, *scn_spec*, ...]

  Specifying the file format retains sections that match one or more *scn_spec* from files matching the *file_spec*. For example, this code retains .intvec sections from all input files:

  ```
  --retain='*(.int*)'
  ```

  You can specify --retain='*(*)' to retain all sections from all input files. However, this does not prevent sections from library members from being optimized out.

- **--retain=** *ar_spec*<*mem_spec*, [*mem_spec*, ...>(*scn_spec*[, *scn_spec*, ...]

  Specifying the archive format retains sections matching one or more *scn_spec* from members matching one or more *mem_spec* from archive files matching *ar_spec*. For example, this code retains the .text sections from printf.c.obj in the rts2800_ml_eabi.lib library:

  ```
  --retain=rts2800_ml_eabi.lib<printf.c.obj>(.text)
  ```

  If the library is specified with the --library option (--library=rts2800_ml_eabi.lib) the library search path is used to search for the library. You cannot specify '*<*>(*)'.

8.4.28 Create an Absolute Listing File (--run_abs Option)

The --run_abs option produces an output file for each file linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

8.4.29 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.

8.4.30 Define Stack Size (--stack_size Option)

The TMS320C28x 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 words 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 words. This example defines a 4K word stack:

```
cl2000 --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 (for COFF) or __TI_STACK_SIZE (for EABI), and assigns it a value equal to the size of the section. The default software stack size is 1K words. See Section 8.6.1 for information about referring to linker symbols in C/C++ code.

8.4.31 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.

8.4.32 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'
```

The --symbol_map option is now supported even if --opt_level=4 was used when compiling.

8.4.33 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 rts2800_ml.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 rts2800_ml.lib for the member that defines symtab and to link in the member.

```
cl2000 --run_linker --undef_sym=symtab file1.c.obj file2.c.obj rts2800_ml.lib
```

If you do not use --undef_sym, this member is not included, because there is no explicit reference to it in file1.c.obj or file2.c.obj.

8.4.34 Display a Message When an Undefined Output Section Is Created (--warn_sections)

In a linker command file, you can set up a SECTIONS directive that describes how input sections are combined into output sections. However, if the linker encounters one or more input sections that do not have a corresponding output section defined in the SECTIONS directive, the linker combines the input sections that have the same name into an output section with that name. By default, the linker does not display a message to tell you that this occurred.

You can use the --warn_sections option to cause the linker to display a message when it creates a new output section.

For more information about the SECTIONS directive, see Section 8.5.5. For more information about the default actions of the linker, see Section 8.7.

8.4.35 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.

8.4.36 Zero Initialization (--zero_init Option)

The C and C++ standards require that global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler supports preinitialization of uninitialized variables by default. To turn this off, specify the linker option --zero_init=off. COFF ABI does not support zero initialization.

The syntax for the --zero_init option is:

--zero_init[={*on*|*off*}]

Disabling Zero Initialization Not Recommended

NOTE: In general, this option it is not recommended. If you turn off zero initialization, automatic initialization of uninitialized global and static objects to zero will not occur. You are then expected to initialize these variables to zero in some other manner.

8.5 Linker Command Files

Linker command files allow you to put linker options and directives in a file; this is useful when you invoke the linker often with the same options and directives. 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 8.5.4). The SECTIONS directive controls how sections are built and allocated (see Section 8.5.5.)
- Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the cl2000 --run_linker command and follow it with the name of the command file:

cl2000 --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 8-1 shows a sample linker command file called link.cmd.

Example 8-1. Linker Command File

```
a.c.obj                    /*  First input filename          */
b.c.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 8-1 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:

```
cl2000 --run_linker link.cmd
```

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

```
cl2000 --run_linker --relocatable link.cmd x.c.obj y.c.obj
```

The linker processes the command file as soon as it encounters the filename, so a.c.obj and b.c.obj are linked into the output module before x.c.obj and y.c.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:

```
cl2000 --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 8-2 shows a sample command file that contains linker directives.

Example 8-2. 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 = 0x7000    length = 0x1000
}

SECTIONS                    /* SECTIONS directive    */
{
  .text:  > SLOW_MEM
  .data:  > SLOW_MEM
  .ebss:  > FAST_MEM
}
```

For more information, see Section 8.5.4 for the MEMORY directive, and Section 8.5.5 for the SECTIONS directive.

8.5.1 Reserved Names in Linker Command Files

The following names (in both uppercase and lowercase) are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

ADDRESS_MASK	END	LENGTH	ORG	SIZE
ALGORITHM	f	LOAD	ORIGIN	START
ALIGN	FILL	LOAD_END	PAGE	TABLE
ATTR	GROUP	LOAD_SIZE	PALIGN	TYPE
BLOCK	HAMMING_MASK	LOAD_START	PARITY_MASK	UNION
COMPRESSION	HIGH	MEMORY	RUN	UNORDERED
COPY	INPUT_PAGE	MIRRORING	RUN_END	VFILL
CRC_TABLE	INPUT_RANGE	NOINIT	RUN_SIZE	
DSECT	l (lowercase L)	NOLOAD	RUN_START	
ECC	LEN	o	SECTIONS	

In addition, any section names used by the TI tools are reserved from being used as the prefix for other names, unless the section will be a subsection of the section name used by the TI tools. For example, section names may not begin with .debug.

8.5.2 Constants in Linker Command Files

You can specify constants with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants (but not binary constants) used in the assembler (see Section 4.7) 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

8.5.3 Accessing Files and Libraries from a Linker Command File

Many applications use custom linker command files (or LCFs) to control the placement of code and data in target memory. For example, you may want to place a specific data object from a specific file into a specific location in target memory. This is simple to do using the available LCF syntax to reference the desired object file or library. However, a problem that many developers run into when they try to do this is a linker generated "file not found" error when accessing an object file or library from inside the LCF that has been specified earlier in the command-line invocation of the linker. Most often, this error occurs because the syntax used to access the file on the linker command-line does not match the syntax that is used to access the same file in the LCF.

Consider a simple example. Imagine that you have an application that requires a table of constants called "app_coeffs" to be defined in a memory area called "DDR". Assume also that the "app_coeffs" data object is defined in a .data section that resides in an object file, app_coeffs.c.obj. The app_coeffs.c.obj file is then included in the object file library app_data.lib. In your LCF, you can control the placement of the "app_coeffs" data object as follows:

```
SECTIONS
{
   ...
   .coeffs: { app_data.lib<app_coeffs.c.obj>(.data) } > DDR
   ...
}
```

Now assume that the app_data.lib object library resides in a sub-directory called "lib" relative to where you are building the application. In order to gain access to app_data.lib from the build command-line, you can use a combination of the –i and –l options to set up a directory search path which the linker can use to find the app_data.lib library:

```
%> cl2000 <compile options/files> -z -i ./lib -l app_data.lib mylnk.cmd <link options/files>
```

The –i option adds the lib sub-directory to the directory search path and the –l option instructs the linker to look through the directories in the directory search path to find the app_data.lib library. However, if you do not update the reference to app_data.lib in mylnk.cmd, the linker will fail to find the app_data.lib library and generate a "file not found" error. The reason is that when the linker encounters the reference to app_data.lib inside the SECTIONS directive, there is no –l option preceding the reference. Therefore, the linker tries to open app_data.lib in the current working directory.

In essence, the linker has a few different ways of opening files:

- If there is a path specified, the linker will look for the file in the specified location. For an absolute path, the linker will try to open the file in the specified directory. For a relative path, the linker will follow the specified path starting from the current working directory and try to open the file at that location.
- If there is no path specified, the linker will try to open the file in the current working directory.
- If a –l option precedes the file reference, then the linker will try to find and open the referenced file in one of the directories in the directory search path. The directory search path is set up via –i options and environment variables (like C_DIR and).

As long as a file is referenced in a consistent manner on the command line and throughout any applicable LCFs, the linker will be able to find and open your object files and libraries.

Returning to the earlier example, you can insert a –l option in front of the reference to app_data.lib in mylnk.cmd to ensure that the linker will find and open the app_data.lib library when the application is built:

```
SECTIONS
{
   ...
   .coeffs: { -l app_data.lib<app_coeffs.c.obj>(.data) } > DDR
   ...
}
```

Another benefit to using the –l option when referencing a file from within an LCF is that if the location of the referenced file changes, you can modify the directory search path to incorporate the new location of the file (using –i option on the command line, for example) without having to modify the LCF.

8.5.4 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 TMS320C28x 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 2.5.

8.5.4.1 Default Memory Model

If you do not use the MEMORY directive, the linker uses a default memory model that is based on the TMS320C28x architecture. For more information about the default memory model, see Section 8.7.

8.5.4.2 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

TMS320C28x devices have separate memory spaces (pages) that occupy the same address ranges (overlay). In the default memory map, one space is dedicated to the program area, while a second is dedicated to the data area. (For detailed information about overlaying pages, see Section 8.5.5.2.7.)

In the linker command file, you configure the address spaces separately by using the MEMORY directive's PAGE option. The linker treats each page as a separate memory space. The TMS320C28x supports up to 255 address spaces, but the number of address spaces available depends on the customized configuration of your device (see the *TMS320C2xx User's Guide* for more information.)

When you use the MEMORY directive, be sure to identify all memory ranges that are available for the program to access at run time. Memory defined by the MEMORY directive is configured; any memory that you do not explicitly account for with MEMORY is unconfigured. The linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in Example 8-3 defines a system that has 4K words of slow external memory at address 0x0000 0C00 in program memory, 32 words of fast external memory at address 0x0000 0060 in data memory, and 512 words of slow external memory at address 0x0000 0200 in data memory. It also demonstrates the use of memory range expressions as well as start/end/size address operators (see Example 8-4).

Example 8-3. The MEMORY Directive

```
/*************************************************************************/
/*                 Sample command file with MEMORY directive             */
/*************************************************************************/
file1.c.obj    file2.c.obj                                  /*  Input files  */
--output_file=prog.out                                  /*  Options      */
#define BUFFER 0

MEMORY
{
   PAGE 0:  PROG:    origin = 0x00000C00, length = 0x00001000 + BUFFER

   PAGE 1:  SCRATCH: origin = 0x00000060, length = 0x00000020
            RAM1:    origin = end(SCRATCH,1) + 0x00000180, length = 0x00000200
}
```

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 32 767 pages. Usually, PAGE 0 specifies program memory, and PAGE 1 specifies data memory. Each PAGE represents a completely independent address space. Configured memory on PAGE 0 can overlap configured memory on PAGE 1 and so on. If you do not specify PAGE for a memory space, the linker defaults to PAGE 0. If you do not specify PAGE in your *allocation* (see Section 8.5.5), the linker allocates initialized sections to PAGE 0 and uninitialized sections to PAGE 1.

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 a 32-bit integer constant expression, 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 a 22-bit integer constant expression, 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 an 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. (See Section 8.5.10.3 for virtual filling of memory ranges when using Error Correcting Code (ECC).)

Filling Memory Ranges

NOTE: If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

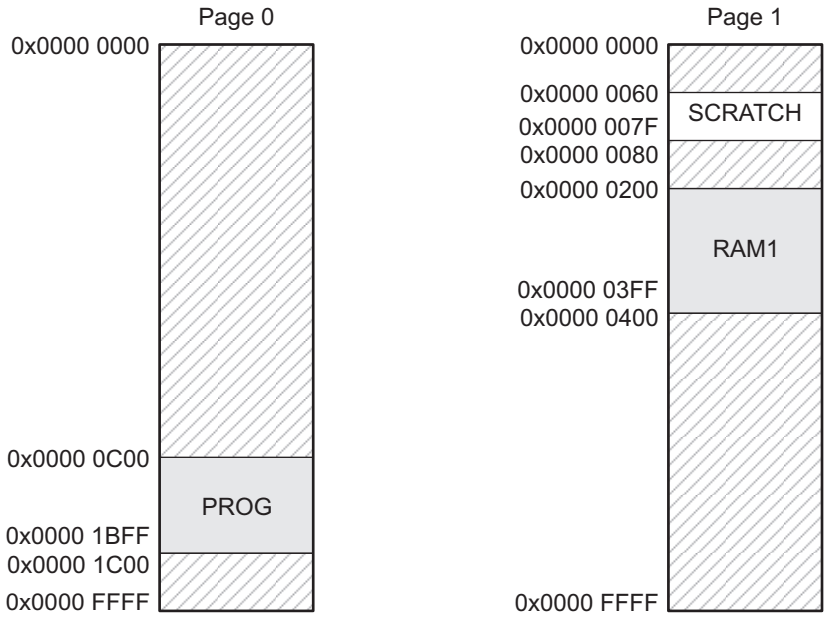
The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFFFFFh:

```
MEMORY
{
    RFILE (RW) : o = 0x00000020, l = 0x00001000, f = 0xFFFFFFFF
}
```

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control placement of output sections. For more information about the SECTIONS directive, see Section 8.5.5.

Figure 8-2 illustrates the memory map shown in Example 8-3

Figure 8-2. Memory Map Defined in Example 8-3



8.5.4.3 Expressions and Address Operators

Memory range origin and length can use expressions of integer constants with the following 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 address operators reference 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 8-4. Origin and Length as Expressions

```
/*****************************************************************************/
/*                    Sample command file with MEMORY directive                */
/*****************************************************************************/
file1.c.obj file2.c.obj                                    /*    Input files     */
--output_file=prog.out                                   /*    Options         */
#define ORIGIN 0x00000000
#define BUFFER 0x00000200
#define CACHE  0x0001000

MEMORY
{
   PAGE 1: FAST_MEM (RX): origin = ORIGIN + CACHE length = 0x00001000 + BUFFER
   PAGE 0: SLOW_MEM (RW): origin = end(FAST_MEM)  length = 0x00001800 - size(FAST_MEM)
   PAGE 0: EXT_MEM  (RX): origin = 0x03000000     length = size(FAST_MEM) - CACHE
```

8.5.5 The SECTIONS Directive

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 RAM1 and allocate the .ebss section into the area named PROG.

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
- Allows you to control where output sections are placed in memory in relation to each other and to the entire memory space (Note that the memory placement order is *not* simply the sequence in which sections occur in the SECTIONS directive.)
- Permits renaming of output sections

For more information, see Section 2.5, Section 2.7, and Section 2.4.6. 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 8.7 describes this algorithm in detail.

8.5.5.1 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.) Section names can refer to sections, subsections, or archive library members. (See Section 8.5.5.4 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. See Section 3.5, Section 3.1.1, and Section 8.5.6.
  Syntax: **load =** *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. See Section 8.5.5.3.
  Syntax: **{** *input_sections* **}**
- **Section type** defines flags for special section types. See Section 8.5.9.
  Syntax: **type = COPY** or
  type = DSECT or
  type = NOLOAD
- **Fill value** defines the value used to fill uninitialized holes. See Section 8.5.12.
  Syntax: **fill =** *value*

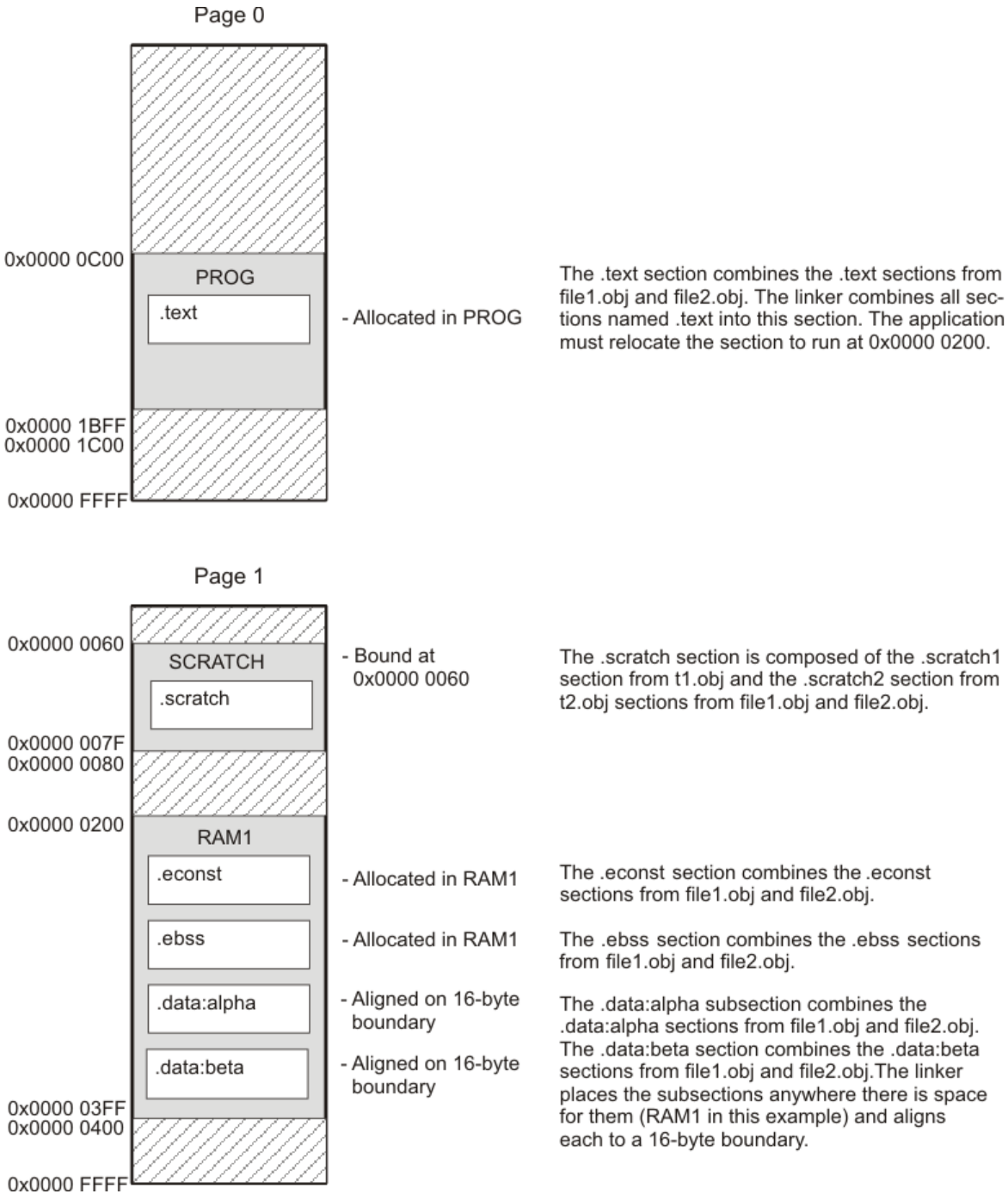
Example 8-5 shows a SECTIONS directive in a sample linker command file. (This example uses COFF section names.)

Example 8-5. The SECTIONS Directive

```
/************************************************/
/*  Sample command file with SECTIONS directive   */
/************************************************/
file1.c.obj    file2.c.obj           /* Input files   */
--output_file=prog.out           /* Options       */

SECTIONS
{
   .text:       load = PROG,   PAGE = 0,
                run  = 0x0200, PAGE = 1
   .econst:     load = RAM1
   .ebss:       load = RAM1
   .scratch:    load = 0x0060, PAGE = 1
      {
         t1.c.obj(.scratch1)
         t2.c.obj(.scratch2)
         endscratch = .;
      }
   .data:alpha: align = 16
   .data:beta:  align = 16
}
```

Figure 8-3 shows the output sections defined by the SECTIONS directive in Example 8-5 (.vectors, .text, .econst, .ebss, .data:alpha, and .data:beta) and shows how these sections are allocated in memory using the MEMORY directive given in Example 8-3.

Figure 8-3. Section Placement Defined by Example 8-5



8.5.5.2 Section Allocation and Placement

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 placement. For more information about using separate load and run placement, see Section 8.5.6.

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to place the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default placement for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control placement 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 placement are separate, all parameters following the keyword LOAD apply to load placement, and those following the keyword RUN apply to run placement. 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 the section must start on an address boundary. `.text: align = 0x100`
Blocking	uses the block keyword to specify the section must fit between two address aligned to the blocking factor. If a section is too large, it starts on an address boundary. `.text: block(0x100)`
Page	specifies the memory page to be used (see Section 8.5.8). If PAGE is not specified, the linker allocates initialized sections to PAGE 0 (program memory) and uninitialized sections to PAGE 1 (data memory). `.text: load = SLOW_MEM PAGE 1`

For the load (usually the only) allocation, 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 8.5.5.3.

Additional information about controlling the order in which code and data are placed in memory is provided in the FAQ topic on section placement.

8.5.5.2.1 Example: Placing Functions in RAM

The --ramfunc compiler option and ramfunc function attribute allow the compiler to specify that a function is to be placed in and executed from RAM. Most newer TI linker command files support the ramfunc option and function attribute by placing such functions in the .TI.ramfunc section. If you see a linker error related to this section, you should add the .TI.ramfunc section to your SECTIONS directive as follows. In these examples, RAM and FLASH are names of MEMORY regions for RAM and Flash memory; the names may be different in your linker command file.

For RAM-based devices:

```
.TI.ramfunc : {} > RAM
```

For Flash-based devices:

```
.TI.ramfunc : {} load=FLASH, run=RAM, table(BINIT)
```

See the Placing functions in RAM wiki page for details.

8.5.5.2.2 Binding

You can set the 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 22-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.

8.5.5.2.3 Named Memory

You can allocate a section into a memory range that is defined by the MEMORY directive (see Section 8.5.4). 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)
   .ebss  :   > FAST_MEM
}
```

In this example, the linker places .text into the area called SLOW_MEM. The .data and .ebss 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    */
   .ebss: > (RW)  /* .ebss --> 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 .ebss 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.

8.5.5.2.4 Controlling Placement 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. You might use the HIGH location specifier in order to keep RTS code separate from application code, so that small changes in the application do not cause large changes to the memory map.

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
{
    .ebss    : {} > RAM
    .esysmem : {} > RAM
    .stack   : {} > RAM (HIGH)
}
```

The HIGH specifier used on the .stack section placement causes the linker to attempt to allocate .stack into the higher addresses within the RAM memory range. The .ebss and .esysmem sections are allocated into the lower addresses within RAM. Example 8-6 illustrates a portion of a map file that shows where the given sections are allocated within RAM for a typical program.

Example 8-6. Linker Placement With the HIGH Specifier

```
.ebss      0    00000200    00000270     UNINITIALIZED
                  00000200    0000011a     rtsxxx.lib  : defs.c.obj (.ebss)
                  0000031a    00000088                 : trgdrv.c.obj (.ebss)
                  000003a2    00000078                 : lowlev.c.obj (.ebss)
                  0000041a    00000046                 : exit.c.obj (.ebss)
                  00000460    00000008                 : memory.c.obj (.ebss)
                  00000468    00000004                 : _lock.c.obj (.ebss)
                  0000046c    00000002                 : fopen.c.obj (.ebss)
                  0000046e    00000002     hello.c.obj (.ebss)

.esysmem   0    00000470    00000120     UNINITIALIZED
                  00000470    00000004     rtsxxx .lib : memory.c.obj (.esysmem)

.stack     0    000008c0    00000140     UNINITIALIZED
                  000008c0    00000002     rtsxxx .lib : boot.c.obj (.stack)
```

As shown in Example 8-6 , the .ebss and .esysmem 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 8-7

The HIGH specifier is ignored if it is used with specific address binding or automatic section splitting (>> operator).

Example 8-7. Linker Placement Without HIGH Specifier

```
.ebss      0    00000200    00000270     UNINITIALIZED
                  00000200    0000011a     rtsxxx.lib  : defs.c.obj (.ebss)
                  0000031a    00000088                 : trgdrv.c.obj (.ebss)
                  000003a2    00000078                 : lowlev.c.obj (.ebss)
                  0000041a    00000046                 : exit.c.obj (.ebss)
                  00000460    00000008                 : memory.c.obj (.ebss)
                  00000468    00000004                 : _lock.c.obj (.ebss)
                  0000046c    00000002                 : fopen.c.obj (.ebss)
                  0000046e    00000002     hello.c.obj (.ebss)

.stack     0    00000470    00000140     UNINITIALIZED
                  00000470    00000002     rtsxxx.lib  : boot.c.obj (.stack)

.esysmem   0    000005b0    00000120     UNINITIALIZED
                  000005b0    00000004     rtsxxx.lib  : memory.c.obj (.esysmem)
```

8.5.5.2.5 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)
```

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 .ebss so that the entire section is contained in a single 128-byte page or begins on that boundary:

```
ebss: 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.

8.5.5.2.6 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 = 0xffff {} > PMEM
```

In this example, the length of the .mytext section is 3 16-bit bytes before the palign operator is applied. The contents of .mytext are as follows:

```
addr content
---- -------
0001 0x1234
0002 0x1234
0003 0x1234
```

After the palign operator is applied, the length of .mytext is 8 bytes, and its contents are as follows:

```
addr content
---- -------
0001 0x1234
0002 0x1234
0003 0x1234
0004 0xffff
0005 0xffff
0006 0xffff
0007 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. 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
---- -------
0001 0x1234
0002 0x1234
0003 0x1234
0004 0x00ff
0005 0x00ff
0006 0x00ff
0007 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
```

8.5.5.2.7 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
   .ebss:   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 .ebss or .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.

8.5.5.3 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 8-8 shows the most common type of section specification; note that no input sections are listed.

Example 8-8. The Most Common Method of Specifying Section Contents

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

In Example 8-8, 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 .ebss or .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. If the filename is hyphenated (or contains special characters), enclose it within quotes:

```
SECTIONS
{
  .text :              /* Build .text output section        */
  {
    f1.c.obj(.text)      /* Link .text section from f1.c.obj    */
    f2.c.obj(sec1)       /* Link sec1 section from f2.c.obj     */
    "f3-new.c.obj"       /* Link ALL sections from f3-new.c.obj */
    f4.c.obj(.text,sec2) /* Link .text and sec2 from f4.c.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.c.obj(sec2).

The specifications in Example 8-8 are actually a shorthand method for the following:

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

The specification *(.text) means *the unallocated .text sections from all input files.* This format is useful if:

- 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.c.obj(xqt)
                    *(.text)
             }
   .data  :  {
                    *(.data)
                    fil.c.obj(table)
             }
}
```

In this example, the .text output section contains a named section xqt from file abc.c.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.c.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.

Each input section acts as a prefix and gathers longer-named sections. For example, the pattern *(.data) matches .dataspecial. This mechanism enables the use of subsections, which are described in the following section.

8.5.5.4 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 linker command file specifying a base name, such as A, selects the section A as well as any subsections of A, such as A:B or A:C:D.

A name such as A:B can 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 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 linker command file or in the input sections supplied to the linker contain multiple colon characters, some change in behavior could be possible. You should carefully consider the impact of the rules for multiple levels to see if it affects a particular system link.

8.5.5.5 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 8-9. Archive Members to Output Sections

```
SECTIONS
{
        boot    >        BOOT1
        {
                -l rtsXX.lib<boot.c.obj> (.text)
                -l rtsXX.lib<exit.c.obj strcpy.c.obj> (.text)
        }

        .rts    >        BOOT2
        {
                -l rtsXX.lib (.text)
        }

        .text    >        RAM
        {
                * (.text)
        }
}
```

In Example 8-9, the .text sections of boot.c.obj, exit.c.obj, and strcpy.c.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 8-9 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 rts2800_ml.lib into the .rtstest section:

```
SECTIONS
{
   .rtstest { -l rts2800_ml.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.

8.5.5.6 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 linker command file, you can let the linker move the section into one of the other areas.

8.5.5.7 Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges

The linker can split output sections among multiple memory ranges for efficient allocation. Use the >> operator to indicate that an output section can be split, if necessary, into the specified memory ranges:

```
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.c.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.c.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
- 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.

8.5.6 Placing a Section at Different Load and Run Addresses

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 3.5 for an overview on run-time relocation.

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 8.8.4.1.)

8.5.6.1 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. See Section 3.1.1 for an overview of load and run addresses.

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 8.5.7.2.)

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 that follow specify load and run addresses.

In this example, align applies only to load:

```
.data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

The following example uses parentheses, but has effects that are identical to the previous example:

```
.data: load = (SLOW_MEM align 32), run = FAST_MEM
```

The following example aligns FAST_MEM to 32 bits for run allocations and aligns all load allocations to 16 bits:

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

For more information on run-time relocation see Section 3.5.

Uninitialized sections (such as .ebss or .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:

```
.ebss: 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 .ebss section is allocated in FAST_MEM.

```
.ebss: load = FAST_MEM
.ebss: run = FAST_MEM
.ebss: > FAST_MEM
```

8.5.6.2 Referring to the Load Address by Using the .label Directive

Normally, any reference to a symbol 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 8-10 and Example 8-11 show the use of the .label directive to copy a section from its load address in SLOW_MEM to its run address in FAST_MEM. Figure 8-4 illustrates the run-time execution of Example 8-10.

If you use the table operator, the .label directive is not needed. See Section 8.8.4.1.

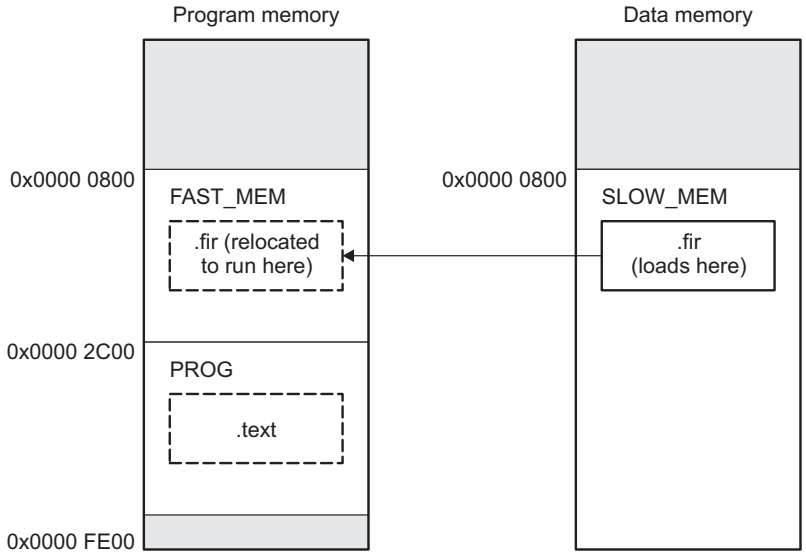
Example 8-10. Moving a Function from Slow to Fast Memory at Run Time

```
;-------------------------------------------------------
;  define a section to be copied from SLOW_MEM to FAST_MEM
;-------------------------------------------------------
    .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 SLOW_MEM to FAST_MEM

;-------------------------------------------------------
    .text

   MOV   XAR6, fir_src
   MOV   XAR7, #fir
   RPT  #(fir_end - fir_src - 1)

k  PWRITE *XAR7, *XAR6++

;-------------------------------------------------------
;  jump to section, now in FAST_MEM
;-------------------------------------------------------
     B  fir
```

Example 8-11. Linker Command File for Example 8-10

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

MEMORY
{
   PAGE 0 :  FAST_MEM  :  origin = 0x00000800,  length = 0x00002400
   PAGE 0 :  PROG      :  origin = 0x00002C00,  length = 0x0000D200
   PAGE 1 :  SLOW_MEM  :  origin = 0x00000800,  length = 0x0000F800
}

SECTIONS
{
   .text: load = PROG PAGE 0
   .fir:  load = SLOW_MEM PAGE 1, run = FAST_MEM PAGE 0
}
```

Figure 8-4. Run-Time Execution of Example 8-10



See Section 8.6.1 for information about referring to linker symbols in C/C++ code.

8.5.7 Using GROUP and UNION Statements

Two SECTIONS statements allow you to organize or conserve memory: GROUP and UNION. Grouping sections causes the linker to allocate them contiguously in memory. Unioning sections causes the linker to allocate them to the same run address.

8.5.7.1 Grouping Output Sections Together

The SECTIONS directive's GROUP option forces several output sections to be allocated contiguously and in the order listed, unless the UNORDERED operator is used. 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 8-12. Allocate Sections Together

```
SECTIONS
{
   .text           /* Normal output section             */
   .ebss           /* 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.

8.5.7.2 Overlaying Sections With the UNION Statement

For some applications, you may want to allocate more than one section that occupies the same address during run time. For example, you may have several routines you want in fast external memory at different 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 8-13, which uses COFF section names, the .ebss sections from file1.c.obj and file2.c.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 8-13. The UNION Statement

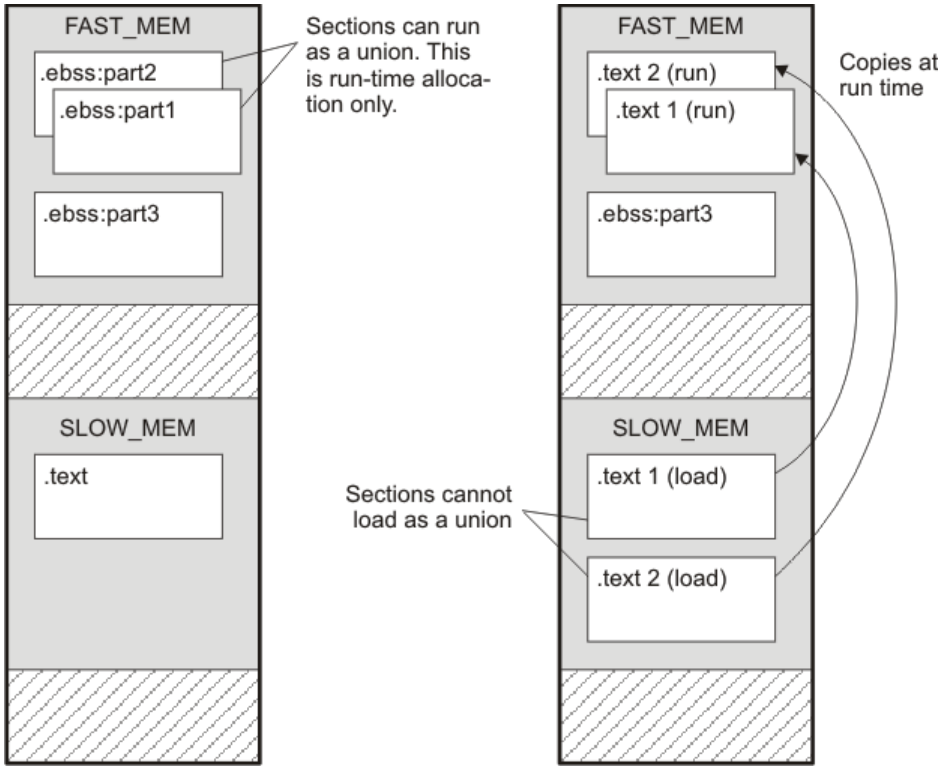
```
SECTIONS
{
   .text: load = SLOW_MEM
   UNION: run  = FAST_MEM
   {
      .ebss:part1: { file1.c.obj(.ebss) }
      .ebss:part2: { file2.c.obj(.ebss) }
   }
      .ebss:part3: run = FAST_MEM { globals.c.obj(.ebss) }
}
```

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 8-14. (There is an exception to this rule when combining an initialized section with uninitialized sections; see Section 8.5.7.3.)

Example 8-14. Separate Load Addresses for UNION Sections

```
UNION run = FAST_MEM
{
   .text:part1: load = SLOW_MEM, { file1.c.obj(.text) }
   .text:part2: load = SLOW_MEM, { file2.c.obj(.text) }
}
```

Figure 8-5. Memory Allocation Shown in Example 8-13 and Example 8-14



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 8.5.8) 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.

8.5.7.3 Using Memory for Multiple Purposes

One way to reduce an application's memory requirement is to use the same range of memory for multiple purposes. You can first use a range of memory for system initialization and startup. Once that phase is complete, the same memory can be repurposed as a collection of uninitialized data variables or a heap. To implement this scheme, use the following variation of the UNION statement to allow one section to be initialized and the remaining sections to be uninitialized.

Generally, an initialized section (one with raw data, such as .text) in a union must have its load allocation specified separately. However, one and only one initialized section in a union can be allocated at the union's run address. By listing it in the UNION statement with no load allocation at all, it will use the union's run address as its own load address.

For example:

```
UNION run = FAST_MEM
{ .cinit .bss }
```

In this example, the .cinit section is an initialized section. It will be loaded into FAST_MEM at the run address of the union. In contrast, .bss is an uninitialized section. Its run address will also be that of the union.

8.5.7.4 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 8-15 shows how two overlays can be grouped together.

Example 8-15. 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*

where *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.

8.5.7.5 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.

8.5.7.6 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)
{
   .ebss    :{}
   .esysmem :{}
   .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)
   {
      .econst :{}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
```

8.5.8 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 8.5.7.2) 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.

8.5.8.1 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).

8.5.8.2 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 8-16 shows how you use the MEMORY directive to obtain this configuration:

Example 8-16. 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 8-16 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.

8.5.8.3 Using Overlay Pages With the SECTIONS Directive

Assume that you are using the MEMORY directive as shown in Example 8-16. 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 8-17 shows how to use the SECTIONS directive overlays to accomplish these objectives.

Example 8-17. SECTIONS Directive Definition for Overlays in Example 7-10

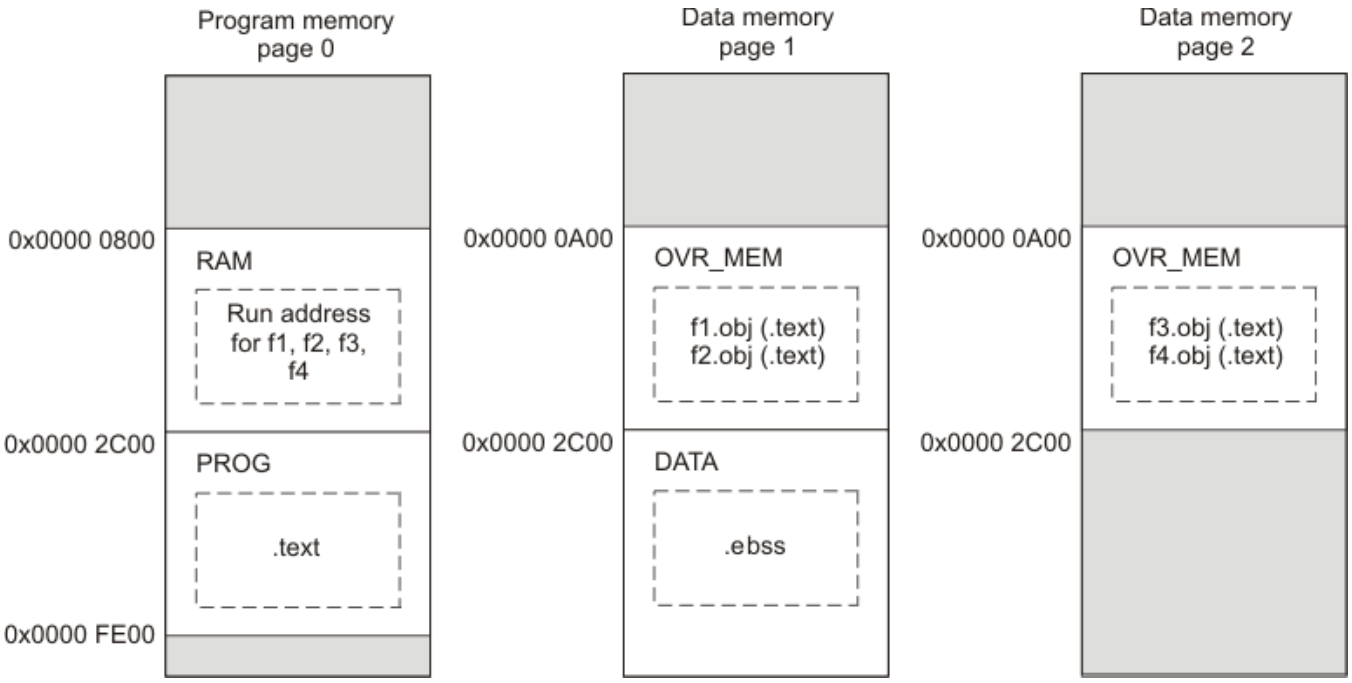
```
SECTIONS
{
   UNION :  run = RAM
   {
      S1 :  load = OVR_MEM PAGE 1
      {
         s1_load = 0x00000A00h;
         s1_start = .;
         f1.c.obj (.text)
         f2.c.obj (.text)
         s1_length = . - s1_start;
      }
      S2 :  load = OVR_MEM PAGE 2
      {
         s2_load = 0x00000A00h;
         s2_start = .;
         f3.c.obj (.text)
         f4.c.obj (.text)
         s2_length = . - s2_start;
      }
   }

   .text: load = PROG PAGE 0
   .data: load = PROG PAGE 0
   .ebss: 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.

8.5.8.4 Memory Allocation for Overlaid Pages

Figure 8-6 shows overlay pages defined by the MEMORY directive in Example 8-16 and the SECTIONS directive in Example 8-17.

Figure 8-6. Overlay Pages Defined in Example 8-16 and Example 8-17



8.5.9 Special Section Types (DSECT, COPY, NOLOAD, and NOINIT)

You can assign the following special types to output sections: DSECT, COPY, NOLOAD, and NOINIT. The NOINIT type is supported for EABI only. 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.c.obj}
   sec2: load = 0x00004000, type =  COPY   {f2.c.obj}
   sec3: load = 0x00006000, type =  NOLOAD {f3.c.obj}
   sec4: load = 0x00008000, type =  NOINIT {f4.c.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.c.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 TMS320C28x 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.
- A NOINIT section is not C auto-initialized by the linker. It is your responsibility to initialize this section as needed. (EABI only)

8.5.10 Configuring Error Correcting Code (ECC) with the Linker

Error Correcting Codes (ECC) can be generated and placed in separate sections through the linker command file. ECC uses extra bits to allow errors to be detected and/or corrected by a device. To enable ECC generation, you must include **--ecc=on** as a linker option on the command line. By default ECC generation is off, even if the ECC directive and ECC specifiers are used in the linker command file. This allows you to fully configure ECC in the linker command file while still being able to quickly turn the code generation on and off via the command line.

The ECC support provided by the linker is compatible with the ECC support in TI Flash memory on various TI devices. TI Flash memory uses a modified Hamming(72,64) code, which uses 8 parity bits for every 64 bits. Check the documentation for your Flash memory to see if ECC is supported. (ECC for read-write memory is handled completely in hardware at run time.)

You can control the details of ECC generation using the ECC specifier in the memory map (Section 8.5.10.1) and the ECC directive (Section 8.5.10.2).

See Section 8.4.12 for command-line options that introduce bit errors into code that has a corresponding ECC section or into the ECC parity bits themselves. Use these options to test ECC error handling code.

ECC can be generated during linking. The ECC data is included in the resulting object file, alongside code and data, as a data section located at the appropriate address. No extra ECC generation step is required after compilation, and the ECC can be uploaded to the device along with everything else.

8.5.10.1 Using the ECC Specifier in the Memory Map

To generate ECC, add a separate memory range to your memory map to hold ECC data and to indicate which memory range contains the Flash data that corresponds to this ECC data. If you have multiple memory ranges for Flash data, you should add a separate ECC memory range for each Flash data range.

The definition of an ECC memory range can also provide parameters for how to generate the ECC data.

The memory map for a device supporting Flash ECC may look something like this:

```
MEMORY {
   VECTORS  : origin=0x00000000 length=0x000020
   FLASH0   : origin=0x00000020 length=0x17FFE0
   FLASH1   : origin=0x00180000 length=0x180000
   STACKS   : origin=0x08000000 length=0x000500
   RAM      : origin=0x08000500 length=0x03FB00
   ECC_VEC  : origin=0xf0400000 length=0x000004 ECC={ input_range=VECTORS }
   ECC_FLA0 : origin=0xf0400004 length=0x02FFFC ECC={ input_range=FLASH0  }
   ECC_FLA1 : origin=0xf0430000 length=0x030000 ECC={ input_range=FLASH1  }
}
```

The specification syntax for ECC memory ranges is as follows:

```
MEMORY {
   <memory specifier1> : <memory attributes> [ vfill=<fill value> ]
   <memory specifier2> : <memory attributes> ECC = {
       input_range = <memory specifier1>
       [ input_page  = <integer>        ]
       [ algorithm   = <algorithm name> ]
       [ fill        = [ true, false ]  ]
   }
}
```

The "ECC" specifier attached to the ECC memory ranges indicates the data memory range that the ECC range covers. The ECC specifier supports the following parameters:

input_range = <*range*>	The data memory range covered by this ECC data range. Required.
input_page = <*page number*>	The page number of the input range. Required only if the input range's name is ambiguous.
algorithm = <*ECC alg name*>	The name of an ECC algorithm defined later in the command file using the ECC directive. Optional if only one algorithm is defined. (See Section 8.5.10.2.)

fill = true | false	Whether to generate ECC data for holes in the initialized data of the input range. The default is "true". Using fill=false produces behavior similar to the nowECC tool. The input range can be filled normally or using a virtual fill (see Section 8.5.10.3).

8.5.10.2 Using the ECC Directive

In addition to specifying ECC memory ranges in the memory map, the linker command file must specify parameters for the algorithm that generates ECC data. You might need multiple ECC algorithm specifications if you have multiple Flash devices.

Each TI device supporting Flash ECC has exactly one set of valid values for these parameters. The linker command files provided with Code Composer Studio include the ECC parameters necessary for ECC support on the Flash memory accessible by the device. Documentation is provided here for completeness.

You specify algorithm parameters with the top-level ECC directive in the linker command file. The specification syntax is as follows:

```
ECC {
   <algorithm name> : parity_mask  = <8-bit integer>
                      mirroring    = [ F021, F035 ]
                      address_mask = <32-bit mask>
}
```

For example:

```
MEMORY {
   FLASH0 : origin=0x00000020 length=0x17FFE0
   ECC_FLA0 : origin=0xf0400004 length=0x02FFFC ECC={ input_range=FLASH0 algorithm=F021 }
}

ECC { F021 : parity_mask = 0xfc
             mirroring = F021 }
```

This ECC directive accepts the following attributes:

algorithm_name	Specify the name you would like to use for referencing the algorithm.
address_mask = <*32-bit mask*>	This mask determines which bits of the address of each 64-bit piece of memory are used in the calculation of the ECC byte for that memory. Default is 0xffffffff, so that all bits of the address are used. (Note that the ECC algorithm itself ignores the lowest bits, which are always zero for a correctly-aligned input block.)
parity_mask = <*8-bit mask*>	This mask determines which ECC bits encode even parity and which bits encode odd parity. Default is 0, meaning that all bits encode even parity.
mirroring = F021 | F035	This setting determines the order of the ECC bytes and their duplication pattern for redundancy. Default is F021.

8.5.10.3 Using the VFILL Specifier in the Memory Map

Normally, specifying a fill value for a MEMORY range creates initialized data sections to cover any previously uninitialized areas of memory. To generate ECC data for an entire memory range, the linker either needs to have initialized data in the entire range, or needs to know what value uninitialized memory areas will have at run time.

In cases where you want to generate ECC for an entire memory range, but do not want to initialize the entire range by specifying a fill value, you can use the "vfill" specifier instead of a "fill" specifier to virtually fill the range:

```
MEMORY {
   FLASH : origin=0x0000  length=0x4000  vfill=0xffffffff
}
```

The vfill specifier is functionally equivalent to omitting a fill specifier, except that it allows ECC data to be generated for areas of the input memory range that remain uninitialized. This has the benefit of reducing the size of the resulting object file.

The vfill specifier has no effect other than in ECC data generation. It cannot be specified along with a fill specifier, since that would introduce ambiguity.

If fill is specified in the ECC specifier, but vfill is not specified, vfill defaults to 0xff.

8.5.11 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. See Section 8.6.1 for information about referring to linker symbols in C/C++ code.

8.5.11.1 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 8.5.11.3. 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.c.obj           /* Input file                          */
cur_tab = Table1;  /* Assign cur_tab to one of the tables */
```

8.5.11.2 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 8.5.5.)

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 = .;}
   .ebss:    {}
}
```

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 8.5.12.

8.5.11.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in Table 8-11.
- 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 8-11 in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in Table 8-11, 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 8-11. Groups of Operators Used in Expressions (Precedence)

Group 1 (Highest Precedence)		Group 6	
! ~ -	Logical NOT Bitwise NOT Negation	&	Bitwise AND
Group 2		**Group 7**	
* / %	Multiplication Division Modulus	|	Bitwise OR
Group 3		**Group 8**	
+ -	Addition Subtraction	&&	Logical AND
Group 4		**Group 9**	
>> <<	Arithmetic right shift Arithmetic left shift	||	Logical OR
Group 5		**Group 10 (Lowest Precedence)**	
== ! = > < < = > =	Equal to Not equal to Greater than Less than Less than or equal to Greater than or equal to	= + = - = * = / =	Assignment A + = B is equivalent to A = A + B A - = B is equivalent to A = A - B A * = B is equivalent to A = A * B A / = B is equivalent to A = A / B

8.5.11.4 Symbols Automatically Defined by the Linker

For the COFF ABI, the linker automatically defines symbols for those sections 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. 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.)
.ebss	is assigned the first address of the .ebss output section. (It marks the *beginning* of uninitialized data.)
end	is assigned the first address following the .ebss output section. (It marks the *end* of uninitialized data.)

The linker automatically defines the following symbols for C/C++ support when the --ram_model or --rom_model option is used.

__TI_STACK_SIZE	is assigned the size of the .stack section. (EABI)
__TI_STACK_END	is assigned the end of the .stack section. (EABI)
__TI_SYSMEM_SIZE	is assigned the size of the .sysmem section. (EABI)
__STACK_SIZE	is assigned the size of the .stack section. (COFF)
__STACK_END	is assigned the end of the .stack section. (COFF)
__SYSMEM_SIZE	is assigned the size of the .esysmem section. (COFF)

These linker-defined symbols can be accessed in any assembly language module if they are declared with a .global directive (see Identify Global Symbols).

See Section 8.6.1 for information about referring to linker symbols in C/C++ code.

8.5.11.5 Assigning Exact Start, End, and Size Values of a Section to a Symbol

The code generation tools currently support the ability to load program code in one area of (slow) memory and run it in another (faster) area. This is done by specifying separate load and run addresses for an output section or group in the linker command file. Then execute a sequence of instructions (the copying code in Example 8-10) 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 in Example 8-10.

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.

8.5.11.6 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.c.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.c.obj(.text)
        end_of_s2 = .;
}
```

This statement creates three symbols:

- end_of_s1—the end address of .text in s1.c.obj
- start_of_s2—the start address of .text in s2.c.obj
- end_of_s2—the end address of .text in s2.c.obj

Suppose there is padding between s1.c.obj and s2.c.obj created as a result of alignment. Then start_of_s2 is not really the start address of the .text section in s2.c.obj, but it is the address before the padding needed to align the .text section in s2.c.obj. This is due to the linker's interpretation of the dot operator as the current PC. It is also true because 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; }
}
```

8.5.11.7 Address and Dimension Operators

Six operators allow you to define symbols for load-time and run-time addresses and sizes:

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.

These 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.

These symbols defined by the linker can be accessed at runtime using the _symval operator, which is essentially a cast operation. For example, suppose your linker command file contains the following:

```
.text: RUN_START(text_run_start), RUN_SIZE(text_run_size) { *(.text) }
```

Your C program can access these symbols as follows:

```
extern char text_run_start, text_run_size;

printf(".text load start is %lx\n", _symval(&text_run_start));
printf(".text load size is %lx\n",  _symval(&text_run_size));
```

See Section 8.6.1 for more information about referring to linker symbols in C/C++ code.

8.5.11.7.1 Input Items

Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.c.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.c.obj(.text)
        end_of_s2 = .;
}
```

This can be rewritten using the START and END operators as follows:

```
outsect:
{
        s1.c.obj(.text) { END(end_of_s1) }
        s2.c.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.

8.5.11.7.2 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.

8.5.11.7.3 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.

8.5.11.7.4 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.c.obj(.text) }
        .text2: load = ROM, SIZE(text2_size) { f2.c.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.

8.5.12 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.

8.5.12.1 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 .ebss or .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.

8.5.12.2 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 8.5.4.2.

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 8.5.11.

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

```
SECTIONS
{
   outsect:
   {
      file1.c.obj(.text)
     . += 0x0100    /* Create a hole with size 0x0100 */
      file2.c.obj(.text)
    . = align(16);  /* Create a hole to align the SPC */
      file3.c.obj(.text)
    }
}
```

The output section outsect is built as follows:

1. The .text section from file1.c.obj is linked in.
2. The linker creates a 256-byte hole.
3. The .text section from file2.c.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.c.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.c.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.c.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.c.obj(.text)
     file1.c.obj(.ebss)           /* This becomes a hole */
   }
}
```

Because the .text section has raw data, all of outsect must also contain raw data. Therefore, the uninitialized .ebss 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.

8.5.12.3 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.c.obj(.text)
      file2.c.obj(.ebss)= 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.c.obj(.text)
      file1.c.obj(.ebss)              /* 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 8.4.14). 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:

```
cl2000 --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.

8.5.12.4 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
{
   .ebss: fill = 0x1234 /* Fills .ebss 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.

8.6 Linker Symbols

This section provides information about using and resolving linker symbols.

8.6.1 Using Linker Symbols in C/C++ Applications

Linker symbols have a name and a value. The value is a 32-bit unsigned integer, even if it represents a pointer value on a target that has pointers smaller than 32 bits.

The most common kind of symbol is generated by the compiler for each function and variable. The value represents the target address where that function or variable is located. When you refer to the symbol by name in the linker command file or in an assembly file, you get that 32-bit integer value.

However, in C and C++ names mean something different. If you have a variable named x that contains the value Y, and you use the name "x" in your C program, you are actually referring to the contents of variable x. If "x" is used on the right-hand side of an expression, the compiler fetches the value Y. To realize this variable, the compiler generates a linker symbol named x with the value &x. Even though the C/C++ variable and the linker symbol have the same name, they don't represent the same thing. In C, x is a variable name with the address &x and content Y. For linker symbols, x is an address, and that address contains the value Y.

Because of this difference, there are some tricks to referring to linker symbols in C code. The basic technique is to cause the compiler to create a "fake" C variable or function and take its address. The details differ depending on the type of linker symbol.

Linker symbols that represent a function address: In C code, declare the function as an extern function. Then, refer to the value of the linker symbol using the same name. This works because function pointers "decay" to their address value when used without adornment. For example:

```
extern void _c_int00(void);

printf("_c_int00 %lx\n", (unsigned long)&_c_int00);
```

Suppose your linker command file defines the following linker symbol:

```
func_sym=printf+100;
```

Your C application can refer to this symbol as follows:

```
extern void func_sym(void);

printf("func_sym %lx\n", _symval(&func_sym));  /* these two are equivalent */
printf("func_sym %lx\n", (unsigned long)&func_sym);
```

Linker symbols that represent a data address: In C code, declare the variable as an extern variable. Then, refer to the value of the linker symbol using the & operator. Because the variable is at a valid data address, we know that a data pointer can represent the value.

Suppose your linker command file defines the following linker symbols:

```
data_sym=.data+100;
xyz=12345
```

Your C application can refer to these symbols as follows:

```
extern char data_sym;
extern int xyz;

printf("data_sym %lx\n", _symval(&data_sym));  /* these two are equivalent */
printf("data_sym %p\n", &data_sym);

myvar = &xyz;
```

Linker symbols for an arbitrary address: In C code, declare this as an extern symbol. The type does not matter. If you are using GCC extensions, declare it as "extern void". If you are not using GCC extensions, declare it as "extern char". Then, refer to the value of the linker symbol mySymbol as _symval(&mySymbol). You must use the _symval operator, which is equivalent to a cast, because the 32-bit value of the linker symbol could be wider than a data pointer. The compiler treats _symval(&mySymbol) in a special way that can represent all 32 bits, even when pointers are 16 bits. Targets that have 32-bit pointers can usually use &mySymbol instead of the _symval operator. However, the portable way to access such linker symbols across TI targets is to use _symval(&mySymbol).

Suppose your linker command file defines the following linker symbol:

```
abs_sym=0x12345678;
```

Your C application can refer to this symbol as follows:

```
extern char abs_sym;

printf("abs_sym %lx\n", _symval(&abs_sym));
```

8.6.2 Declaring Weak Symbols

NOTE: Weak symbols are supported only in EABI mode.

In a linker command file, an assignment expression outside a MEMORY or SECTIONS directive can be used to define a linker-defined symbol. To define a weak symbol in a linker command file, use the "weak" operator in an assignment expression to designate that the symbol as eligible for removal from the output file's symbol table if it is not referenced. For example, you can define "ext_addr_sym" as follows:

```
weak(ext_addr_sym) = 0x12345678;
```

When the linker command file is used to perform the final link, then "ext_addr_sym" is presented to the linker as a weak absolute symbol; it will not be included in the resulting output file if the symbol is not referenced.

See Section 2.6.3 for details about how weak symbols are handled by the linker.

8.6.3 Resolving Symbols with 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 7.1 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 8.4.17.3). 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.c.obj and f2.c.obj both reference an external function named *clrscr*.
- Input file f1.c.obj references the symbol *origin*.
- Input file f2.c.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:

```
cl2000 --run_linker f1.c.obj f2.c.obj liba.lib libc.lib
```

then:

- Member 1 of liba.lib satisfies the f1.c.obj and f2.c.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:

```
cl2000 --run_linker f1.c.obj f2.c.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 8.4.33.) The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
cl2000 --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 8.5.5.

Section 8.4.17 describes methods for specifying directories that contain object libraries.

8.7 Default Placement Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections you choose *not* to specify must still be handled by the linker. The linker uses algorithms to build and allocate sections in coordination with any specifications you do supply.

If you do not use the MEMORY and SECTIONS directives, the linker allocates output sections as though the memory map and section definitions were as shown in Example 8-18 were specified.

Example 8-18. Default Allocation for TMS320C28x Devices

```
MEMORY
{
   PAGE 0: PROG:  origin = 0x000040  length = 0x3fffc0
   PAGE 1: DATA:  origin = 0x000000  length = 0x010000
   PAGE 1: DATA1: origin = 0x010000  length = 0x3f0000

}
SECTIONS
{
   .text:    PAGE = 0
   .data:    PAGE = 0
   .cinit:   PAGE = 0   /* Used only for C programs */
   .ebss:    PAGE = 1
}
```

See Section 2.5.1 for information about default memory allocation.

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 this default allocation. Instead, allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described next in Section 8.7.1.

8.7.1 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 8.5.5 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.c.obj and f2.c.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 8.4.34) 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 8-18. (See Section 8.5.4 for more information on configuring memory.)

8.7.2 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 supply 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.

If you want to control the order in which code and data are placed in memory, see the FAQ topic on section placement.

8.8 Using Linker-Generated Copy Tables

The linker supports extensions to the linker command file syntax that enable the following:

- Make it easier for you to copy objects from load-space to run-space at boot time
- Make it easier for you to manage memory overlays at run time
- Allow you to split GROUPs and output sections that have separate load and run addresses

For an introduction to copy tables and their use, see Section 3.3.3.

8.8.1 Using Copy Tables for Boot Loading

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 to develop such an application 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 to on-chip memory at boot time:

- The load location (load page id and address)
- The run location (load 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.

8.8.2 Using Built-in Link Operators in Copy Tables

You can avoid some of this maintenance burden by using the LOAD_START(), RUN_START(), and SIZE() operators that are already part of the linker command file syntax . For example, instead of building the application to generate a .map file, the linker command file can be annotated:

```
SECTIONS
{
   .flashcode: { app_tasks.c.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 8.8.1.

While maintenance of the copy table is reduced markedly, you must still carry the burden of keeping the copy table contents in sync with the symbols that are defined in the linker command file. Ideally, the linker would generate the boot copy table automatically. This would avoid having to build the application twice *and* free you from having to explicitly manage the contents of the boot copy table.

For more information on the LOAD_START(), RUN_START(), and SIZE() operators, see Section 8.5.11.7.

8.8.3 Overlay Management Example

Consider an application that contains a memory overlay that must be managed at run time. The memory overlay is defined using a UNION in the linker command file as illustrated in Example 8-19:

Example 8-19. Using a UNION for Memory Overlay

```
SECTIONS
{
   ...
   UNION
   {
      GROUP
      {
         .task1: { task1.c.obj(.text) }
         .task2: { task2.c.obj(.text) }

      } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)

      GROUP
      {
         .task3: { task3.c.obj(.text) }
         .task4: { task4.c.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.

8.8.4 Generating Copy Tables With the table() Operator

The linker supports extensions to the linker command file syntax that enable you to do the following:

- Identify any object components that may need to be copied from load space to run space at some point during the run of an application
- Instruct the linker to automatically generate a copy table that contains (at least) the load address, run address, and size of the component that needs to be copied
- Instruct the linker to generate a symbol specified by you that provides the address of a linker-generated copy table. For instance, Example 8-19 can be written as shown in Example 8-20:

Example 8-20. Produce Address for Linker Generated Copy Table

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1: { task1.c.obj(.text) }
         .task2: { task2.c.obj(.text) }

      } load = ROM, table(_task12_copy_table)

      GROUP
      {
         .task3: { task3.c.obj(.text) }
         .task4: { task4.c.obj(.text) }

      } load = ROM, table(_task34_copy_table)

   } run = RAM
   ...
}
```

Using the SECTIONS directive from Example 8-20 in the linker command file, the linker generates two copy tables named: _task12_copy_table and _task34_copy_table. Each copy table provides the load address, run address, and size of the GROUP that is associated with the copy table. This information is accessible from application source code using the linker-generated symbols, _task12_copy_table and _task34_copy_table, which provide the addresses of the two copy tables, respectively.

Using this method, you need not 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.

8.8.4.1 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 8.8.4. The table operator can be used with compression; see Section 8.8.5.

8.8.4.2 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. This table is handled before the .cinit section is used to initialize variables at startup. For example, the linker command file for the boot-loaded application described in Section 8.8.2 can be rewritten as follows:

```
SECTIONS
{
   .flashcode: { app_tasks.c.obj(.text) }
   load = FLASH, run = PMEM,
      table(BINIT)
   ...
}
```

For this example, the linker creates a copy table that can be accessed through a special linker-generated symbol, __binit__, which contains the list of all object components that need to be copied from their load location to their run location at boot-time. If a linker command file does not contain any uses of table(BINIT), then the __binit__ symbol is given a value of -1 to indicate that a boot-time copy table does not exist for a particular application.

You can apply the table(BINIT) specification to an output section, GROUP, or UNION member. If used in the context of a UNION, only one member of the UNION can be designated with table(BINIT). If applied to a GROUP, then none of that GROUP's members may be marked with table(BINIT).The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table(BINIT) specification.

8.8.4.3 Using the table() Operator to Manage Object Components

If you have several pieces of code that need to be managed together, then you can apply the same table() operator to several different object components. In addition, if you want to manage a particular object component in multiple ways, you can apply more than one table() operator to it. Consider the linker command file excerpt in Example 8-21:

Example 8-21. Linker Command File to Manage Object Components

```
SECTIONS
{
   UNION
   {
      .first: { a1.c.obj(.text), b1.c.obj(.text), c1.c.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second: { a2.c.obj(.text), b2.c.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.

8.8.4.4 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 8-22 illustrates how you can control the placement of the linker-generated copy table sections using the input section names in the linker command file.

Example 8-22. Controlling the Placement of the Linker-Generated Copy Table Sections

```
SECTIONS
{
   UNION
   {
      .first: { a1.c.obj(.text), b1.c.obj(.text), c1.c.obj(.text) }
             load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second: { a2.c.obj(.text), b2.c.obj(.text) }
             load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra: load = EMEM, run = PMEM, table(BINIT)
   ...
   .ovly: { } > BMEM
   .binit: { } > BMEM
}
```

For the linker command file in Example 8-22, 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.

8.8.4.5 Splitting Object Components and Overlay Management

It is possible to split sections that have separate load and run placement instructions. 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 8-23.

Example 8-23. 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 8-24 illustrates a possible driver for such an application.

Example 8-24. Split Object Component Driver

```
#include <cpy_tbl.h>

extern COPY_TABLE task13_ctbl;
extern 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();
   ...
}
```

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.

8.8.5 Compression

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.

Copy table compression is supported only when you use EABI object modules by specifying the --abi=eabi option. This feature is not supported for COFF object modules.

You can specify compression in two ways:

- The linker command line option --copy_compression=*compression_kind* can be used to apply the specified compression to any output section that has a table() operator applied to it.
- The table() operator accepts an optional compression parameter. The syntax is: .

  table(*name* **, compression=** *compression_kind* **)**

  The *compression_kind* can be one of the following types:

  – **off**. Don't compress the data.
  – **lzss**. Compress data using Lempel-Ziv-Storer-Szymanski compression.
  – **rle**. Compress data using Run Length Encoding.

  A table() operator without the compression keyword uses the compression kind specified using the command line option --copy_compression. If no *compression_kind* was specified with the command-line option, the default is LZSS compression.

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.

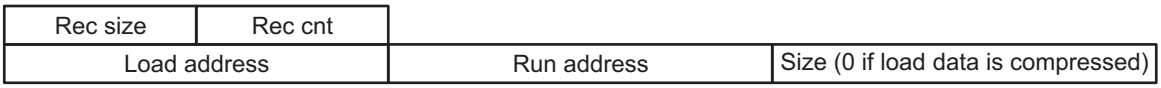
You cannot force the linker to compress the data when doing so does not result in savings.

You cannot compress the decompression routines or any member of a GROUP containing .cinit.

8.8.5.1 Compressed Copy Table Format

The copy table format is the same irrespective of the *compression_kind*. The size field of the copy record is overloaded to support compression. Figure 8-7 illustrates the compressed copy table layout.

Figure 8-7. Compressed Copy Table



In Figure 8-7, 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.

8.8.5.2 Compressed Section Representation in the Object File

The linker creates a separate input section to hold the compressed data. 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 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.

The linker allocates load space for the .task1.load input section in the memory area that was specified for load placement for the .task1 section. There is only a single load section to represent the load placement of .task1 - .task1.load. If the .task1 data had not been compressed, there would be two allocations for the .task1 input section: one for its load placement and another for its run placement.

8.8.5.3 Compressed Data Layout

For C2000, the unit size is 16-bits, which is the size of an unsigned char type on C2000. For compressed data sections on C2000, all data is stored in 16-bit units.

The compressed load data has the following layout:

16-bit index	Compressed data

The first 16 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 8-8.

Figure 8-8. 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().

8.8.5.4 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 16 bits from the load address.
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.

8.8.5.5 Compression Algorithms

The following subsections provide information about decompression algorithms for the RLE and LZSS formats. To see example decompression algorithms, refer to the following functions in the Run-Time Support library:

- **RLE:** The __TI_decompress_rle() function in the copy_decompress_rle.c file.
- **LZSS:** The __TI_decompress_lzss() function in the copy_decompress_lzss.c file.

Additional information about compression algorithms can be found in the *C28x Embedded Application Binary Interface Application Report* (SPRAC71) EABI specification.

Run Length Encoding (RLE):

16-bit index	Initialization data compressed using run length encoding

The data following the 16-bit index is compressed using run length encoded (RLE) format. C2000 uses a simple run length encoding that can be decompressed using the following algorithm. See copy_decompress_rle.c for details.

1. Read the first 16 bits, Delimiter (D).
2. Read the next 16 bits (B).
3. If B != D, copy B to the output buffer and go to step 2.
4. Read the next 16 bits (L).
   a. If L == 0, then length is either a 32 bit value or we've reached the end of the data, read the next 16 bits (L).
      1. If L == 0, the end of the data is reached. Go to the "end of processing" step.
      2. If L !=0, then L is the most significant 16-bits of the 32-bit run length, L.hi. Read the next 16-bits, which is L.lo. Concatenate L.hi with L.lo to form the 32-bit run length.
   b. Else if L > 0 and L < 4, copy D to the output buffer L times. Go to step 2.
   c. Else, length is 16-bit value (L).
5. Read the next 16 bits (C); C is the repeat character.
6. Write C to the output buffer L times; go to step 2.
7. End of processing.

The C2000 run-time support library has a routine __TI_decompress_rle() to decompress data compressed using RLE. The first argument to this function is the address pointing to the 16 bits after the 16-bit index. The second argument is the run address from the C auto initialization record.

Lempel-Ziv-Storer-Szymanski Compression (LZSS):

16-bit index	Data compressed using LZSS

The data following the 8-bit index is compressed using LZSS compression. The C2000 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 16 bits after the 16-bit Index, and the second argument is the run address from the C auto initialization record.

The decompression algorithm for LZSS is as follows. See copy_decompress_lzss.c for details on the LZSS algorithm.

1. Read 16 bits, which are the encoding flags (F) marking the start of the next LZSS encoded packet.
2. For each bit (B) in F, starting from the least significant to the most significant bit, do the following:
   a. If (B & 0x1), read the next 16 bits and write it to the output buffer. Then advance to the next bit (B) in F and repeat this step.
   b. Else read the next 16-bits into temp (T), length (L) = (T & 0xf) + 2, and offset (O) = (T >> 4).
      i. If L == 17, read the next 16-bits (L'); then L += L'.
      ii. If O == LZSS end of data (LZSS_EOD), we've reached the end of the data, and the algorithm is finished.
      iii. At position (P) = output buffer - Offset (O) - 1, read L bytes from position P and write them to the output buffer.
      iv. Go to step 2a.

8.8.6 Copy Table Contents

To use a copy table generated by the linker, you must know the contents of the copy table. This information is included in a run-time-support library header file, cpy_tbl.h, which contains a C source representation of the copy table data structure that is generated by the linker. Example 8-25 shows the copy table header file.

Example 8-25. TMS320C28x cpy_tbl.h File

```
/*****************************************************************************/
/* cpy_tbl.h                                                                 */
/*                                                                           */
/* Specification of copy table data structures which can be automatically   */
/* generated by the linker (using the table() operator in the LCF).         */
/*                                                                           */
/*****************************************************************************/
/* Copy Record Data Structure                                                */
/*****************************************************************************/
typedef struct copy_record
{
   unsigned int          src_pgid;
   unsigned int          dst_pgid;
   unsigned long         src_addr;
   unsigned long         dst_addr;
   unsigned long         size;

} COPY_RECORD;

/*****************************************************************************/
/* Copy Table Data Structure                                                 */
/*****************************************************************************/
typedef struct copy_table
{
   unsigned int          rec_size;
   unsigned int          num_recs;
   COPY_RECORD           recs[1];
} COPY_TABLE;

/*****************************************************************************/
/* Prototype for general purpose copy routine.                               */
/*****************************************************************************/
extern void copy_in(COPY_TABLE *tp);

/*****************************************************************************/
/* Prototypes for utilities used by copy_in() to move code/data between     */
/* program and data memory (see cpy_utils.asm for source).                   */
/*****************************************************************************/
extern void ddcopy(unsigned long src, unsigned long dst);
extern void dpcopy(unsigned long src, unsigned long dst);
extern void pdcopy(unsigned long src, unsigned long dst);
extern void ppcopy(unsigned long src, unsigned long dst);
```

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 8.8.4.2, the .first and .extra output sections will each have their own COPY_RECORD entries in the BINIT copy table. The BINIT copy table will then look like this:

```
COPY_TABLE __binit__ = { 12, 2,
                         { <load page id of .first>,
                           <run page id of .first>,
                           <load address of .first>,
                           <run address of .first>,
                           <size of .first> },
                         { <load page id of .extra>,
                           <run page id of .extra>,
                           <load address of .extra>,
                           <run address of .extra>,
                           <size of .extra> } };
```

8.8.7 General Purpose Copy Routine

The cpy_tbl.h file in Example 8-25 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 8-26.

Example 8-26. Run-Time-Support cpy_tbl.c File

```
/*****************************************************************************/
/* cpy_tbl.c                                                                 */
/*                                                                           */
/* 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>

void copy_in(COPY_TABLE *tp)
{
   unsigned int i;
   for (i = 0; i < tp->num_recs; i++)
   {
      COPY_RECORD *crp = &tp->recs[i];
      unsigned int cpy_type = 0;
      unsigned int j;

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

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

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

8.9 Linker-Generated CRC Tables

The linker supports an extension to the linker command file syntax that enables the verification of code or data by means of Cyclic Redundancy Code (CRC). The linker computes a CRC value for the specified region at link time, and stores that value in target memory such that it is accessible at boot or run time. The application code can then compute the CRC for that region and ensure that the value matches the linker-computed value.

The run-time-support library does not supply a routine to calculate CRC values at boot or run time, however a limited reference implementation in C is provided in Appendix C.

Examples that perform cyclic redundancy checking using linker-generated CRC tables are provided in the Tools Insider blog in TI's E2E community.

8.9.1 The crc_table() Operator

For any section that should be verified with a CRC, the linker command file must be modified to include the crc_table() operator. The specification of a CRC algorithm is optional. The syntax is:

crc_table(*user_specified_table_name***[, algorithm=***xxx***])**

The linker uses the CRC algorithm from any specification given in a crc_table() operator. If that specification is omitted, the CRC32_PRIME algorithm is used. The linker includes CRC table information in the map file. This includes the CRC value as well as the algorithm used for the calculation.

The CRC table generated for a particular crc_table() instance can be accessed through the table name provided as an argument to the crc_table() operator. The linker creates a symbol with this name and assigns the address of the CRC table as the value of the symbol. The CRC table can then be accessed from the application using the linker-generated symbol.

The crc_table() operator can be applied to an output section, a GROUP, a GROUP member, a UNION, or a UNION member. If applied to a GROUP or UNION, the operator is applied to each member of the GROUP or UNION.

You can include calls in your application to a routine that will verify CRC values for relevant sections. You must provide this routine. See below for more details on the data structures and suggested interface.

8.9.2 Restrictions

It is important to note that the CRC generator used by the linker is parameterized as described in the crc_tbl.h header file (see Example 8-31). Any CRC calculation routine employed outside of the linker must function in the same way to ensure matching CRC values. The linker cannot detect a mismatch in the parameters. To understand these parameters, see *A Painless Guide to CRC Error Detection Algorithms* by Ross Williams, which is likely located at http://www.ross.net/crc/download/crc_v3.txt.

Only the CRC algorithm names and identifiers specified in crc_tbl.h are supported. All other names and ID values are reserved for future use. Your system may not include built-in hardware that computes all these CRC algorithms. Consult the documentation for your hardware for more detail. The following CRC algorithms are supported:

- CRC8_PRIME
- CRC16_ALT
- CRC16_802_15_4
- CRC_CCITT
- CRC24_FLEXRAY
- CRC32_PRIME
- CRC32_C

- CRC64_ISO

The supported CRC algorithms are specified by published standards, including the Powerline Related Intelligent Metering Evolution (PRIME) standard and IEEE 802.15.4. The Viterbi, Complex Math and CRC Unit (VCU) module available on some C28x devices provides efficient instructions for CRC calculation using these algorithms. You might want to take advantage of the VCU module to compute the CRC at run time. For details, see the VCU module documentation in *TMS320x28xx, 28xxx DSP Peripherals Reference Guide* (SPRU566).

There are also restrictions which will be enforced by the linker:

- CRC can only be requested at final link time.
- CRC can only be applied to initialized sections.
- CRC can be requested for load addresses only.
- Certain restrictions also apply to CRC table names.

8.9.3 Examples

The crc_table() operator is similar in syntax to the table() operator used for copy tables. A few simple examples of linker command files follow.

Example 8-27. Using crc_table() Operator to Compute the CRC Value for .text Data

```
...
SECTIONS
{
   ...
   .section_to_be_verified: {a1.c.obj(.text)} crc_table(_my_crc_table_for_a1)
}
```

Example 8-27 defines a section named ".section_to_be_verified", which contains the .text data from the a1.c.obj file. The crc_table() operator requests that the linker compute the CRC value for the .text data and store that value in a table named "my_crc_table_for_a1". This table will contain all the information needed to invoke a user-supplied CRC calculation routine, and verify that the CRC calculated at run time matches the linker-generated CRC. The table can be accessed from application code using the symbol my_crc_table_for_a1, which should be declared of type "extern CRC_TABLE". This symbol will be defined by the linker. The application code might resemble the following.

```
#include "crc_tbl.h"

extern CRC_TABLE my_crc_table_for_a1;

verify_a1_text_contents()
{
   ...
   /* Verify CRC value for .text sections of a1.c.obj. */
   if (my_check_CRC(&my_crc_table_for_a1)) puts("OK");
}
```

The my_check_CRC() routine is shown in detail in Appendix C.

Example 8-28. Specifying an Algorithm in the crc_table() Operator

```
...
SECTIONS
{
   ...
   .section_to_be_verified_2:   {b1.c.obj(.text)} load=SLOW_MEM, run=FAST_MEM,
       crc_table(_my_crc_table_for_b1, algorithm=CRC8_PRIME)

.TI.crctab:  > CRCMEM
}
...
```

In Example 8-28, the CRC algorithm is specified in the crc_table() operator. The specified algorithm is used to compute the CRC of the text data from b1.c.obj. The CRC tables generated by the linker are created in the special section .TI.crctab, which can be placed in the same manner as other sections. In this case, the CRC table _my_crc_table_for_b1 is created in section .TI.crctab:_my_crc_table_for_b1, and that section is placed in the CRCMEM memory region.

Example 8-29. Using a Single Table for Multiple Sections

```
...
SECTIONS
{
   .section_to_be_verified_1: {a1.c.obj(.text)}
      crc_table(_my_crc_table_for_a1_and_c1)

   .section_to_be_verified_3: {c1.c.obj(.text)}
      crc_table(_my_crc_table_for_a1_and_c1, algorithm=CRC16_802_15_4)
}
...
```

In Example 8-29 the same identifier, _my_crc_table_for_a1_and_c1, is specified for both a1.c.obj and c1.c.obj. The linker creates a single table that contains entries for both text sections. Multiple CRC algorithms can occur in a single table. In this case, _my_crc_table_for_a1_and_c1 contains an entry for the text data from a1.c.obj using the default CRC algorithm, and an entry for the text data from c1.c.obj using the CRC16_802_15_4 algorithm. The order of the entries is unspecified.

Example 8-30. Applying the crc_table() Operator to a GROUP or UNION

```
...
SECTIONS
{
   UNION
   {
      section1: {} crc_table(table1, algorithm=CRC16_ALT)
      section2:
   }  crc_table(table2, algorithm=CRC32_PRIME)
}
```

When the crc_table() operator is applied to a GROUP or a UNION, the linker applies the table specification to the members of the GROUP or UNION.

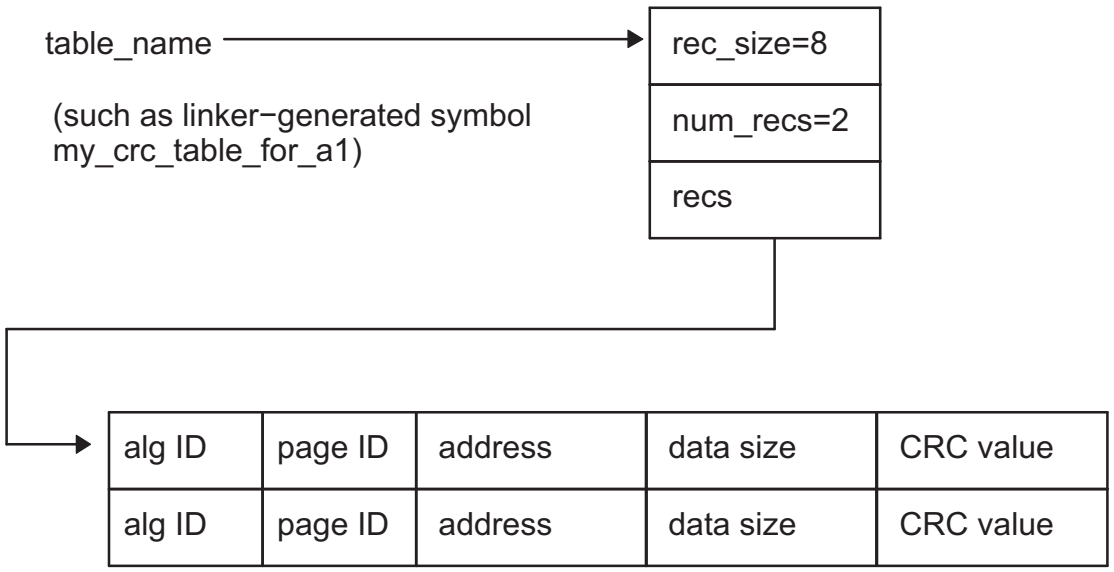
In Example 8-30 the linker creates two CRC tables, table1 and table2. table1 contains one entry for section1, using algorithm CRC16_ALT. Because both sections are members of the UNION, table2 contains entries for section1 and section2, using algorithm CRC32_PRIME. The order of the entries in table2 is unspecified.

8.9.4 Interface

The CRC generation function uses a mechanism similar to the copy table functionality. Using the syntax shown above in the linker command file allows specification of code/data sections that have CRC values computed and stored in the run time image. This section describes the table data structures created by the linker, and how to access this information from application code.

The CRC tables contain entries as detailed in the run-time-support header file crc_tbl.h, as illustrated in Figure 8-9.

Figure 8-9. CRC_TABLE Conceptual Model



The crc_tbl.h header file is included in Example 8-31. This file specifies the C structures created by the linker to manage CRC information. It also includes the specifications of the supported CRC algorithms. A full discussion of CRC algorithms is beyond the scope of this document, and the interested reader should consult the referenced document for a description of the fields shown in the table. The following fields are relevant to this document.

- Name – text identifier of the algorithm, used by the programmer in the linker command file.
- ID – the numeric identifier of the algorithm, stored by the linker in the crc_alg_ID member of each table entry.
- Order – the number of bits used by the CRC calculation.
- Polynomial – used by the CRC computation engine.
- Initial Value – the initial value given to the CRC computation engine.

Example 8-31. The CRC Table Header, crc_tbl.h

```
/*****************************************************************************/
/* crc_tbl.h                                                                 */
/*                                                                           */
/* PRELIMINARY - SUBJECT TO CHANGE                                           */
/*                                                                           */
/* Specification of CRC table data structures which can be automatically     */
/* generated by the linker (using the crc_table() operator in the linker     */
/* command file).                                                            */
/*****************************************************************************/
/*****************************************************************************/
/*                                                                           */
/*  The CRC generator used by the linker is based on concepts from the       */
/*  document:                                                                */
/*     "A Painless Guide to CRC Error Detection Algorithms"                  */
/*                                                                           */
```

Example 8-31. The CRC Table Header, crc_tbl.h (continued)

```
/* Author : Ross Williams (ross@guest.adelaide.edu.au.).                    */
/* Date   : 3 June 1993.                                                    */
/* Status : Public domain (C code).                                         */
/*                                                                          */
/* Description : For more information on the Rocksoft^tm Model CRC          */
/* Algorithm, see the document titled "A Painless Guide to CRC Error        */
/* Detection Algorithms" by Ross Williams (ross@guest.adelaide.edu.au.).    */
/* This document is likely to be in "ftp.adelaide.edu.au/pub/rocksoft" or   */
/* at http:www.ross.net/crc/download/crc_v3.txt.                            */
/*                                                                          */
/* Note: Rocksoft is a trademark of Rocksoft Pty Ltd, Adelaide, Australia.  */
/****************************************************************************/

#include <stdint.h>        /* For uintXX_t */

/****************************************************************************/
/* CRC Algorithm Specifiers                                                 */
/*                                                                          */
/* The following specifications, based on the above cited document, are used */
/* by the linker to generate CRC values.                                    */
/*                                                                          */
/*                                                                          */
/* ID  Name          Order  Polynomial   Initial     Ref Ref  CRC XOR   Zero */
/*                                       Value       In  Out  Value     Pad  */
/*--------------------------------------------------------------------------- */
/* 0, "CRC32_PRIME",    32, 0x04c11db7,  0x00000000,  0,  0, 0x00000000,  1  */
/* 1, "CRC16_802_15_4", 16, 0x00001021,  0x00000000,  0,  0, 0x00000000,  1  */
/* 2, "CRC16_ALT",      16, 0x00008005,  0x00000000,  0,  0, 0x00000000,  1  */
/* 3, "CRC8_PRIME",      8, 0x00000007,  0x00000000,  0,  0, 0x00000000,  1  */
/*                                                                          */
/*                                                                          */
/* Users should specify the name, such as CRC32_PRIME, in the linker command */
/* file.  The resulting CRC_RECORD structure will contain the corresponding  */
/* ID value in the crc_alg_ID field.                                        */
/****************************************************************************/
#define CRC32_PRIME       0     /* Poly = 0x04c11db7 */ /* DEFAULT ALGORITHM */
#define CRC16_802_15_4    1     /* Poly = 0x00001021 */
#define CRC16_ALT         2     /* Poly = 0x00008005 */
#define CRC8_PRIME        3     /* Poly = 0x00000007 */

/*********************************************************/
/* CRC Record Data Structure                             */
/* NOTE: The list of fields and the size of each field   */
/*       varies by target and memory model.              */
/*********************************************************/
typedef struct crc_record
{
uint16_t         crc_alg_ID;    /* CRC algorithm ID             */
uint16_t         page_id;       /* page number of data          */
uint32_t         addr;          /* Starting address             */
uint32_t         size;          /* size of data in 16-bit units */
uint32_t         crc_value;
} CRC_RECORD;

/*********************************************************/
/* CRC Table Data Structure                              */
/*********************************************************/
typedef struct crc_table
{
uint16_t         rec_size;
uint16_t         num_recs;
CRC_RECORD       recs[1];
} CRC_TABLE;
```

In the CRC_TABLE struct, the array recs[1] is dynamically sized by the linker to accommodate the number of records contained in the table (num_recs). A user-supplied routine to verify CRC values should take a table name and check the CRC values for all entries in the table. An outline of such a routine is shown in Appendix C.

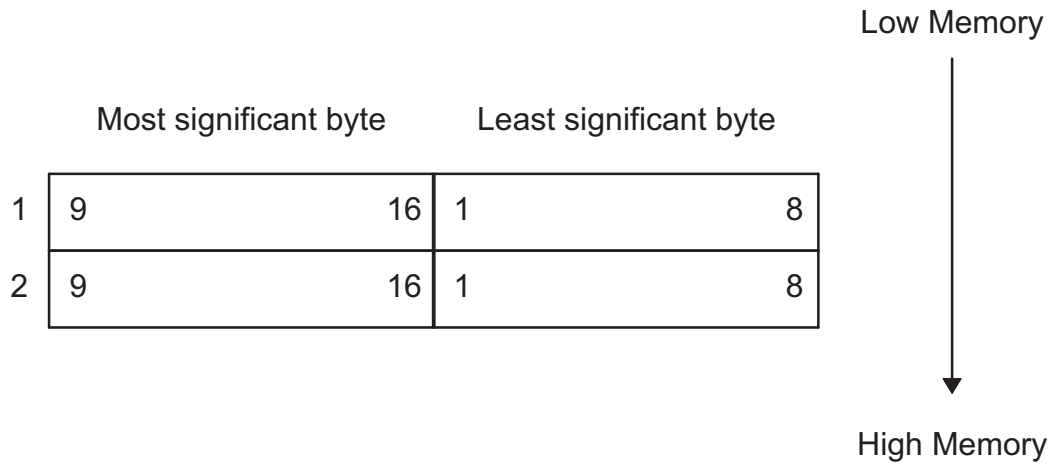
8.9.5 A Special Note Regarding 16-Bit char

C2000 is a 16-bit word addressable target, which means that its char data type is 16 bits. However, CRC algorithms operate on 8-bit units, which we shall call "octets". When computing a CRC on a C2000 section, the data cannot be fed to the CRC loop char-by-char, it must be fed octet-by-octet.

The data needs to be fed to the CRC in the order it would if the C2000 were a 8-bit machine, so we need to consider which of the two octets in the char to feed first. C2000 is a little-endian machine, but it does not make sense to talk about the endianness of the bits in an indivisible unit such as char. By convention, we consider the data in a char to be stored least-significant octet first, then most-significant octet.

Abstractly, the CRC algorithm computes the CRC bit-by-bit in the order the bits appear in the data. For a machine with 8-bit chars, this order is considered to proceed from the MSB through the LSB of each byte starting with byte 0. However, for C2000, the CRC starts with the MSB through LSB of the LEAST significant octet of byte 0, then the MSB through LSB of the MOST significant octet of byte 0, and so on for the rest of the bytes.

Figure 8-10. CRC Data Flow Example



8.10 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 8.4.3.2.)
- 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 8.4.23.)
- 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.

  Since the ELF object file format is used with EABI, input sections are not combined into output sections during a partial link unless a matching SECTIONS directive is specified in the link step command file.
- 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 8.4.18.1).
- 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 8.4.26, Section 3.3.2.1, and Section 3.3.2.2.)

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.

```
cl2000 --run_linker --relocatable --output_file=tempout1 file1.com
```

file1.com contains:

```
SECTIONS
{
    ss1:   {
           f1.c.obj
           f2.c.obj
            .
            .
            .
           fn.c.obj
           }
}
```

Step 2: Link the file file2.com; use the --relocatable option to retain relocation information in the output file tempout2.out.

```
cl2000 --run_linker --relocatable --output_file=tempout2 file2.com
```

file2.com contains:

```
SECTIONS
{
   ss2:   {
           g1.c.obj
           g2.c.obj
            .
            .
            .
           gn.c.obj
           }
}
```

Step 3: Link tempout1.out and tempout2.out.

```
cl2000 --run_linker --map_file=final.map --
output_file=final.out tempout1.out tempout2.out
```

8.11 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:

```
cl2000 --run_linker --rom_model --
output_file prog.out prog1.c.obj prog2.c.obj ... rts2800_ml.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 TMS320C28x C/C++ language, including the run-time environment and run-time-support functions, see the *TMS320C28x Optimizing C/C++ Compiler User's Guide*.

8.11.1 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.c.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.c.obj; referencing _c_int00 ensures that boot.c.obj is automatically linked in from the run-time-support library. When a program begins running, it executes boot.c.obj first. The boot.c.obj symbol contains code and data for initializing the run-time environment and performs the following tasks:

- Sets up the system stack and configuration registers
- Processes the run-time *.cinit* initialization table and autoinitializes global variables (when the linker is invoked with the --rom_model option)
- Disables interrupts and calls _main

The run-time-support object libraries contain boot.c.obj. You can:

- Use the archiver to extract boot.c.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.c.obj when you use the --ram_model or --rom_model option).

8.11.2 Object Libraries and Run-Time Support

The *TMS320C28x 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.

8.11.3 Setting the Size of the Stack and Heap Sections

The C/C++ language uses two uninitialized sections called .esysmem and .stack for the memory pool used by the malloc( ) functions and the run-time stacks, respectively. You can set the size of these by using the --heap_size or --stack_size 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 1K words and the default size of the stack is 1K words.

See Section 8.4.15 for setting heap sizes and Section 8.4.30 for setting stack sizes.

Linking the .stack Section

NOTE: The .stack section must be linked into the low 64K of data memory (PAGE 1) since the SP is a 16-bit register and cannot access memory locations beyond the first 64K.

8.11.4 Initializing and AutoInitialzing 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. See Section 3.3.2.1 for details.

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. See Section 3.3.2.2 for details.

See Section 3.3.2.3 for information about the steps that are performed when you invoke the linker with the --ram_model or --rom_model option.

8.12 Linker Example

This example links three object files named demo.c.obj, ctrl.c.obj, and tables.c.obj and creates a program called demo.out.

Assume that target memory has the following program memory configuration:

Memory Type	Address Range	Contents
Program	0x0f0000 to 0x3fffbf	SLOW_MEM
	0x3fffc0 to 0x3fffff	Interrupt vector table
Data	0x000040 to 0x0001ff	Stack
	0x000200 to 0x0007ff	FAST_MEM_1
	0x3ed000 to 0x3effff	FAST_MEM_2

The output sections are constructed in the following manner:

- Executable code, contained in the .text sections of demo.c.obj, fft.c.obj, and tables.c.obj, is linked into program memory ROM.
- Variables, contained in the var_defs section of demo.c.obj, are linked into data memory in block FAST_MEM_2.
- Tables of coefficients in the .data sections of demo.c.obj, tables.c.obj, and fft.c.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.c.obj, which contains buffers and variables, is linked by default into page 1 of the block STACK, since it is not explicitly linked.

Example 8-32 shows the linker command file for this example. Example 8-33 shows the map file.

Example 8-32. Linker Command File, demo.cmd

```
/*************************************************************/
/***                 Specify Linker Options                ***/
/*************************************************************/
--output_file=demo.out     /* Name the output file             */
--map_file=demo.map        /* Create an output map             */

/*************************************************************/
/***                 Specify the Input Files               ***/
/*************************************************************/

demo.c.obj
fft.c.obj
tables.c.obj

/*************************************************************/
/***             Specify the Memory Configuration          ***/
/*************************************************************/

MEMORY
{
     PAGE 0: SLOW_MEM   (R):   origin=0x3f0000    length=0x00ffc0
             VECTORS    (R):   origin=0x3fffc0    length=0x000040

     PAGE 1: STACK      (RW):  origin=0x000040    length=0x0001c0
             FAST_MEM_1 (RW):  origin=0x000200    length=0x000600
             FAST_MEM_2 (RW):  origin=0x3ed000    length=0x003000
}

/*************************************************************/
/***               Specify the Output Sections             ***/
/*************************************************************/

SECTIONS
{
     vectors    : { } > VECTORS page=0
     .text      : load = SLOW_MEM, page = 0   /* link in .text */

     .data      : fill = 07A1Ch, Load=FAST_MEM_1, page=1
     {
                tables.c.obj(.data)              /* .data input */
                fft.c.obj(.data)                 /* .data input */
                . += 100h;   /* create hole, fill with 0x07A1C */
     }

     var_defs   : { } > FAST_MEM_2 page=1       /* defs in RAM */
     .ebss      : page=1, fill=0x0ffff     /*.ebss fill and link*/
}
/*************************************************************/
/***                   End of Command File                 ***/
/*************************************************************/
```

Invoke the linker by entering the following command:

`cl2000 --run_linker demo.cmd`

This creates the map file shown in Example 8-33 and an output file called demo.out that can be run on a TMS320C28x.

Example 8-33. Output Map File, demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: 0

MEMORY CONFIGURATION
             name        origin     length     attributes     fill
            --------    --------   ---------   ----------   --------
PAGE 0:      SLOW_MEM    003f0000   0000ffc0    R
             VECTORS     003fffc0   00000040    R
PAGE 1:      STACK       00000040   000001c0    RW
             FAST_MEM_1  00000200   00000600    RW
             FAST_MEM_2  003ed000   00003000    RW

SECTION ALLOCATION MAP
 output                                  attributes/
section    page    origin       length      input sections
--------   ----   ----------   ----------   ----------------
vectors     0     003fffc0     00000000     UNINITIALIZED
.text       0     003f0000     0000001a
                  003f0000     0000000e     demo.c.obj (.text)
                  003f000e     00000000     tables.c.obj (.text)
                  003foooe     0000000c     fft.c.obj (.text)
var_defs    1     003ed000     00000002
                  003ed000     00000002     demo.c.obj (var_defs)
.data       1     00000200     0000010c
                  00000200     00000004     tables.c.obj (.data)
                  00000204     00000000     fft.c.obj (.data)
                  00000204     00000100     --HOLE-- [fill = 7a1c]
                  00000304     00000008     demo.c.obj (.data)
.ebss       0     00000040     00000069
                  00000040     00000068     demo.c.obj (.ebss) [fill=ffff]
                  000000a8     00000000     fft.c.obj (.ebss)
                  000000a8     00000001     tables.c.obj (.ebss) [fill=ffff]
xy          1     000000a9     00000014     UNINITIALIZED
                  000000a9     00000014     demo.c.obj (xy)

GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

address    name
--------   ----
00000040   .ebss
00000200   .data
003f0000   .text
00000040   ARRAY
000000a8   TEMP
00000040   ___ebss__
00000200   ___data__
0000030c   ___edata__
000000a9   ___end__
003f001a   ___etext__
003f0000   ___text__
003f000e   _func1
003f0000   _main
0000030c   edata
000000a9   end
003f001a   etext


GLOBAL SYMBOLS: SORTED BY Symbol Address

address    name
--------   ----
00000040   ARRAY
00000040   ___ebss__
```

Example 8-33. Output Map File, demo.map (continued)

```
00000040   .ebss
000000a8   TEMP
000000a9   ___end__
000000a9   end
00000200   ___data__
00000200   .data
0000030c   edata
0000030c   ___edata__
003f0000   _main
003f0000   .text
003f0000   ___text__
003f000e   _func1
003f001a   etext
003f001a   ___etext__

[16 symbols]
```

Absolute Lister Description

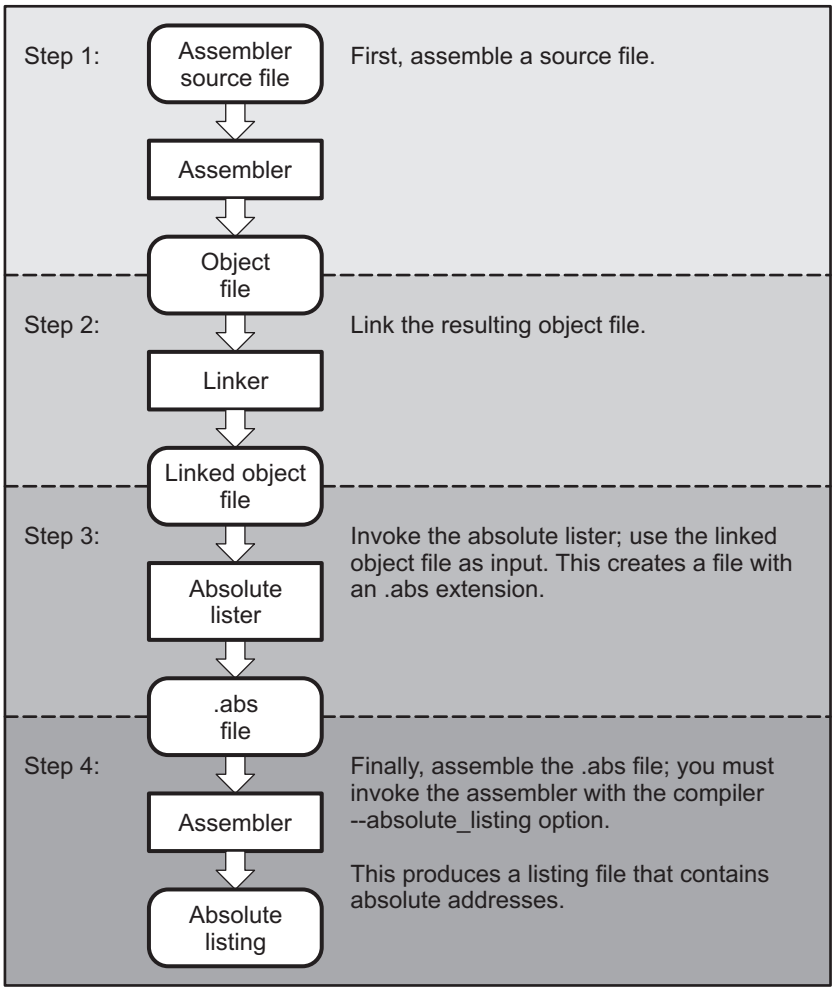
The TMS320C28x absolute lister is a debugging tool that accepts linked object files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code. Manually, this could be a tedious process requiring many operations; however, the absolute lister utility performs these operations automatically.

Topic **Page**

9.1 Producing an Absolute Listing

Figure 9-1 illustrates the steps required to produce an absolute listing.

Figure 9-1. Absolute Lister Development Flow



9.2 Invoking the Absolute Lister

The syntax for invoking the absolute lister is:

abs2000 [*-options*] *input file*

abs2000 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.

-fs specifies a directory for the output files. For example, to place the .abs file generated by the absolute lister in C:\ABSDIR use this command:

```
abs2000 -fs C:\ABSDIR filename.out
```

If the -fs option is not specified, the absolute lister generates the .abs files in the current directory.

-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:

cl2000 --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:

```
abs2000 -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.

9.3 Absolute Lister Example

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

module1.asm

```
        .text
array   .usect  ".ebss",100
dflag   .usect  ".ebss", 2
        .copy   globals.def
        MOV     ACC, #offset
        MOV     ACC, #dflag
```

module2.asm

```
offset  .usect  ".ebss", 2
        .copy   globals.def
        MOV     ACC, #offset
        MOV     ACC, #array
```

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**:

```
cl2000 module1
cl2000 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
{
        PAGE 0:   ROM:  origin=2000h  length=2000h
        PAGE 1:   RAM:  origin=8000h  length=8000h
}
SECTIONS
{
        .data: >RAM
        .text: >ROM
        .ebss: >RAM
}
```

Invoke the linker:

```
cl2000 --run_linker bttest.cmd
```

This command creates an executable object file called bttest.out; use this file as input for the absolute lister.

Step 3: Now, invoke the absolute lister:

```
abs2000 bttest.out
```

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

module1.abs:

```
          .nolist
array     .setsym     000008000h
dflag     .setsym     000008064h
offset    .setsym     000008066h
.data     .setsym     000008000h
edata     .setsym     000008000h
.text     .setsym     000002000h
etext     .setsym     000002008h
.usect    .setsym     000008000h
end       .setsym     000008068h
          .setsect    ".text",000002000h
          .setsect    ".data",000008000h
          .setsect    ".ebss",00008000h
          .list
          .text
          .copy       "module1.asm"
```

module2.abs:

```
          .nolist
 array    .setsym     000008000h
 dflag    .setsym     000008064h
 offset   .setsym     000008066h
 .data    .setsym     000008000h
 edata    .setsym     000008000h
 .text    .setsym     000002000h
 etext    .setsym     000002008h
 .usect   .setsym     000008000h
 end      .setsym     000008068h
          .setsect    ".text",000002004h
          .setsect    ".data",000008000h
          .setsect    ".ebss",00008066h
          .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):

```
cl2000 --absolute_listing module1.abs
cl2000 --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 9-1) and module2.lst (see Example 9-2).

Example 9-1. module1.lst

```
module1.abs                                                     PAGE    1

     15 002000                .text
     16                       .copy       "module1.asm"
      1 002000               .text
      2 008000       array   .usect  ".ebss",100
      3 008064       dflag   .usect  ".ebss",2
      4                      .copy   globals.def
      1                      .global dflag
      2                      .global array
      3                      .global offset
      5 002000 FF20!          MOV     ACC,#offset
        002001 8066
      6 002002 FF20-          MOV     ACC,#dflag
        002003 8064
```

Example 9-2. module2.lst

```
module2.abs                                                     PAGE    1

     15 002004              .text
     16                     .copy       "module2.asm"
      1 008066       offset  .usect  ".ebss",2
      2                      .copy   globals.def
      1                      .global dflag
      2                      .global array
      3                      .global offset
      3 002004 FF20-          MOV     ACC,#offset
        002005 8066
      4 002006 FF20!          MOV     ACC,#array
        002007 8000
```

Cross-Reference Lister Description

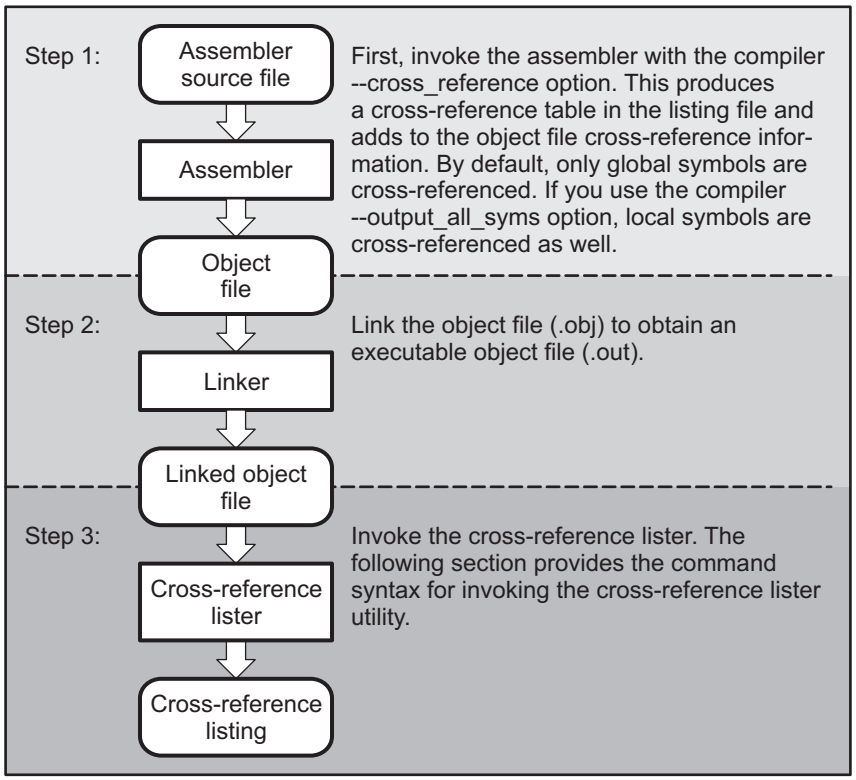
The TMS320C28x cross-reference lister is a debugging tool. This utility accepts linked object files as input and produces a cross-reference listing as output. This listing shows symbols, their definitions, and their references in the linked source files.

Topic **Page**

10.1 Producing a Cross-Reference Listing

Figure 10-1 illustrates the steps required to produce a cross-reference listing.

Figure 10-1. The Cross-Reference Lister Development Flow



10.2 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:

xref2000 [*options*] [*input filename* [*output filename*]]

xref2000 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.

10.3 Cross-Reference Listing Example

These terms defined appear in the cross-reference listing in Example 10-1:

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 10-1 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 10-1. 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 .usect section

Example 10-1 is an example of cross-reference listing.

Example 10-1. Cross-Reference Listing

```
================================================================================
Symbol: _SETUP

Filename      RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________      ____  _________  _________    ______  _______  _______  _______
demo.asm      EDEF  '00000018   00000018       18       13       20
================================================================================
Symbol: _fill_tab

Filename      RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________      ____  _________  _________    ______  _______  _______  _______
ctrl.asm      EDEF  '00000000   00000040       10       5
================================================================================
Symbol: _x42

Filename      RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________      ____  _________  _________    ______  _______  _______  _______
demo.asm      EDEF  '00000000   00000000        7       4        18
================================================================================
Symbol: gvar

Filename      RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________      ____  _________  _________    ______  _______  _______  _______
tables.asm    EDEF  "00000000   08000000       11       10
================================================================================
```

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**

11.1 Invoking the Object File Display Utility

The object file display utility, *ofd2000*, 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:

ofd2000 [*options*] *input filename* [*input filename*]

ofd2000 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.

--call_graph Prints function stack usage and callee information in XML format. While the XML output may be accessed by a developer, this option was primarily designed to be used by tools such as Code Composer Studio to display an application's worst case stack usage.

--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 ofd2000 --dwarf_display=help.

--dwarf Appends DWARF debug information to program output.

--help Displays help

--output=*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", ofd2000 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 ofd2000 --obj_display=help.

--verbose Prints verbose text output.

--xml 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.

11.2 Invoking the Disassembler

The disassembler, *dis2000*, 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:

dis2000 *input filename*[.] [*output filename*]

dis2000	is the command that invokes the disassembler.
input filename[.*ext*]	For EABI, this 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 For COFF, this is a COFF object file (.obj) or an executable file (.out).
output filename	is the name of the optional output file to which the disassembly will be written. If an output filename is not specified, the disassembly is written to standard output.

11.3 Invoking the Name Utility

The name utility, *nm2000*, 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:

nm2000 [-*options*] [*input filenames*]

nm2000	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:	
	--all (-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.
	--prep_fname (-f)	prepends file name to each symbol.
	--global (-g)	prints only global symbols.
	--help (-h)	shows the current help screen.
	--format:long (-l)	produces a detailed listing of the symbol information.
	--sort:value (-n)	sorts symbols numerically rather than alphabetically.

--output (-o) *file*	outputs to the given file.
--sort:none (-p)	causes the name utility to not sort any symbols.
--quiet (-q)	(quiet mode) suppresses the banner and all progress information.
--sort:reverse (-r)	sorts symbols in reverse order.
--dynamic (-s)	lists symbols in the dynamic symbol table for an ELF object module.
-t	also prints tag information symbols for a COFF object module.
--undefined (-u)	only prints undefined symbols.

11.4 Invoking the Strip Utility

The strip utility, *strip2000*, removes symbol table and debugging information from object and executable files. To invoke the strip utility, enter the following:

strip2000 [**-p**] *input filename* [*input filename*]

strip2000	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:

--help (-h)	displays help information.
--outfile (-o) *filename*	writes the stripped output to filename.
--postlink (-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 static executable files.
--rom	Strip readonly sections and segments.

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 TMS320C28x 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:

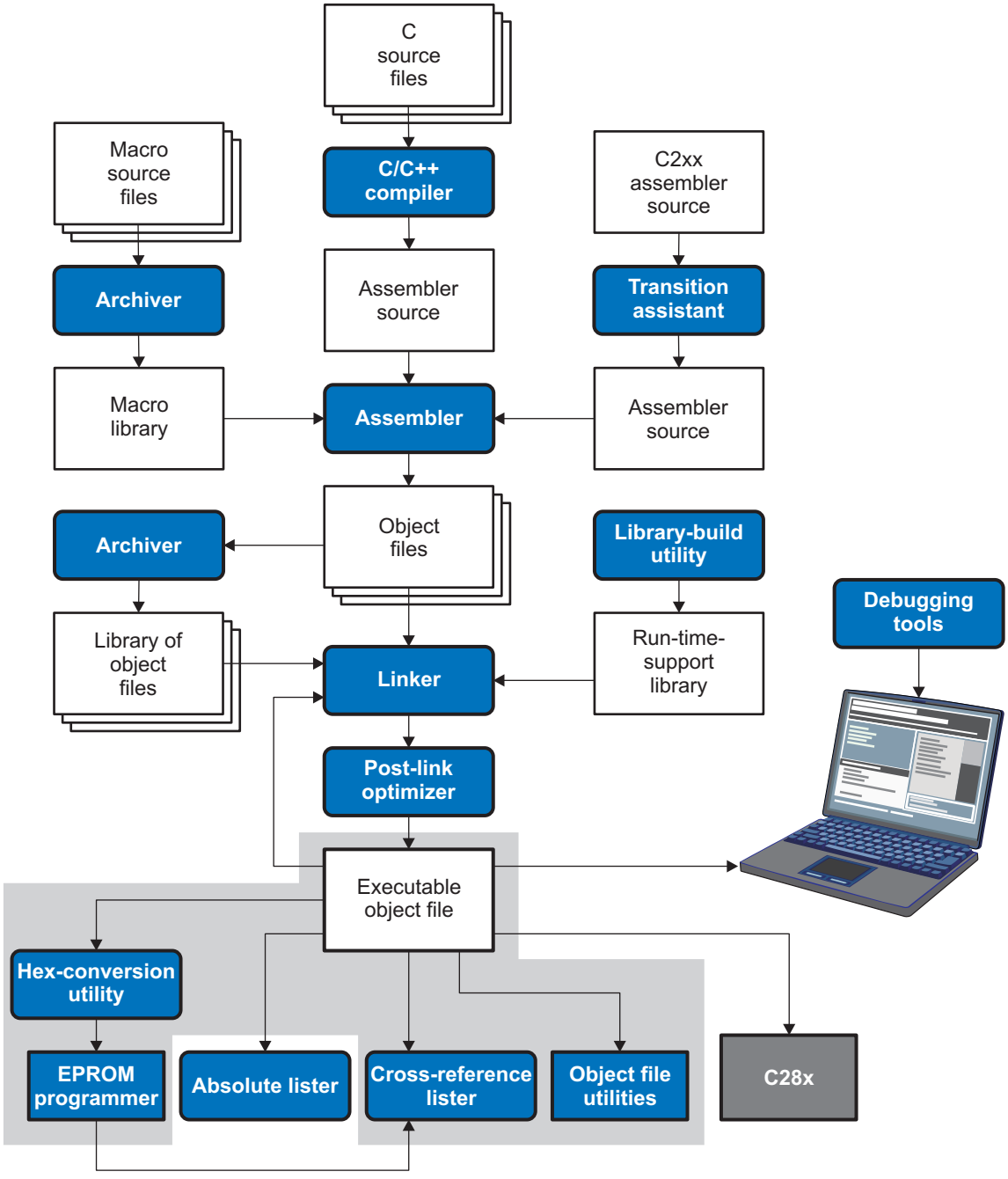
- ASCII-Hex, supporting 16-bit addresses
- Extended Tektronix (Tektronix)
- Intel MCS-86 (Intel)
- Motorola Exorciser (Motorola-S), supporting 16-bit addresses
- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses
- Texas Instruments TI-TXT format, supporting 16-bit addresses
- C arrays

Topic **Page**

12.1 The Hex Conversion Utility's Role in the Software Development Flow

Figure 12-1 highlights the role of the hex conversion utility in the software development process.

Figure 12-1. The Hex Conversion Utility in the TMS320C28x Software Development Flow



12.2 Invoking the Hex Conversion Utility

There are two basic methods for invoking the hex conversion utility:

- **Specify the options and filenames on the command line.** The following example converts the file firmware.out into TI-Tagged format, producing two output files, firm.lsb and firm.msb.

```
hex2000 -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:

```
hex2000 hexutil.cmd
```

In addition to regular command line information, you can use the hex conversion utility ROMS and SECTIONS directives in a command file.

12.2.1 Invoking the Hex Conversion Utility From the Command Line

To invoke the hex conversion utility, enter:

hex2000 [*options*] *filename*

hex2000 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 12-1 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 12.2.2).

Table 12-1. Basic Hex Conversion Utility Options

Option	Alias	Description	See
		General Options	
--byte	-byte	Number output locations by bytes rather than by target addressing	--
--entrypoint=*addr*	-e	Specify the entry point at which to begin execution after boot loading	Table 12-2
--exclude={*fname*(*sname*) | *sname*}	-exclude	If the filename (*fname*) is omitted, all sections matching *sname* will be excluded.	Section 12.7
--fill=*value*	-fill	Fill holes with *value*	Section 12.9.2
--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 12.2.2
--image	-image	Select image mode	Section 12.9.1
--linkerfill	-linkerfill	Include linker fill sections in images	--
--map=*filename*	-map	Generate a map file	Section 12.4.2
--memwidth=*value*	-memwidth	Define the system memory word width (default 16 bits)	Section 12.3.2
--order={LS|MS}	-order	Specify data ordering (endianness)	Section 12.3.4
--outfile=*filename*	-o	Specify an output filename	Section 12.8
--quiet	-q	Run quietly (when used, it must appear *before* other options)	Section 12.2.2

Table 12-1. Basic Hex Conversion Utility Options (continued)

Option	Alias	Description	See
--romwidth=*value*	-romwidth	Specify the ROM device width (default depends on format used). This option is ignored for the TI-TXT and TI-Tagged formats.	Section 12.3.3
--zero	-zero, -z	Reset the address origin to 0 in image mode	Section 12.9.3
Diagnostic Options			
--diag_error=*id*		Categorizes the diagnostic identified by *id* as an error	Section 12.14
--diag_remark=*id*		Categorizes the diagnostic identified by *id* as a remark	Section 12.14
--diag_suppress=*id*		Suppresses the diagnostic identified by *id*	Section 12.14
--diag_warning=*id*		Categorizes the diagnostic identified by *id* as a warning	Section 12.14
--display_error_number		Displays a diagnostic's identifiers along with its text	Section 12.14
--issue_remarks		Issues remarks (nonserious warnings)	Section 12.14
--no_warnings		Suppresses warning diagnostics (errors are still issued)	Section 12.14
--set_error_limit=*count*		Sets the error limit to *count*. The linker abandons linking after this number of errors. (The default is 100.)	Section 12.14
Boot Options			
--boot	-boot	Convert all sections into bootable form (use instead of a SECTIONS directive)	Table 12-2
--bootorg=*addr*	-bootorg	Specify the source address of the boot loader table	Table 12-2
--cmac=file		Specify a file containing the CMAC key for use with secure flash boot on TMS320F2838x devices.	Section 12.12
--gpio8	-gpio8	Specify table source as the GP I/O port, 8-bit mode. (Aliased by --can8)	Table 12-2
--gpio16	-gpio16	Specify table source as the GP I/O port, 16-bit mode	Table 12-2
--lospcp=*value*	-lospcp	Specify the initial value for the LOSPCP register	Table 12-2
--sci8	-sci8	Specify table source as the SCI-A port, 8-bit mode	Table 12-2
--spi8	-spi8	Specify table source as the SPI-A port, 8-bit mode	Table 12-2
--spibrr=*value*	-spibrr	Specify the initial value for the SPIBRR register	Table 12-2
Output Options			
--array		Select array output format	Section 12.10
--ascii	-a	Select ASCII-Hex	Section 12.15.1
--binary	-b	Select binary (Must have memory width of 8 bits.)	--
--intel	-i	Select Intel	Section 12.15.2
--motorola=1	-m1	Select Motorola-S1	Section 12.15.3
--motorola=2	-m2	Select Motorola-S2	Section 12.15.3
--motorola=3	-m3	Select Motorola-S3 (default -m option)	Section 12.15.3
--tektronix	-x	Select Tektronix (default format when no output option is specified)	Section 12.15.4
--ti_tagged	-t	Select TI-Tagged	Section 12.15.5
--ti_txt		Select TI-Txt	Section 12.15.6
Load Image Options			
--load_image		Select load image	Section 12.6
--section_name_prefix=*string*		Specify the section name prefix for load image object files	Section 12.6

12.2.2 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 12.4.)
- **SECTIONS directive.** The hex conversion utility SECTIONS directive specifies which sections from the object file are selected. (See Section 12.5.)
- **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:

hex2000 *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:

```
hex2000 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 */
--outfile=firm.msb   /* output file */
```

  You can invoke the hex conversion utility by entering:

```
hex2000 firmware.cmd
```

- This example shows how to convert a file called appl.out into eight hex files in Intel format. Each output file is one byte wide and 4K bytes long.

```
appl.out          /* input file   */
--intel           /* Intel format */
--map=appl.mxp    /* map file     */

ROMS
{
   ROW1: origin=0x00000000 len=0x4000 romwidth=8
         files={ appl.u0 appl.u1 app1.u2 appl.u3 }
   ROW2: origin=0x00004000 len=0x4000 romwidth=8
         files={ app1.u4 appl.u5 appl.u6 appl.u7 }
}

SECTIONS
{   .text, .data, .cinit, .sect1, .vectors, .econst:
}
```

12.3 Understanding Memory Widths

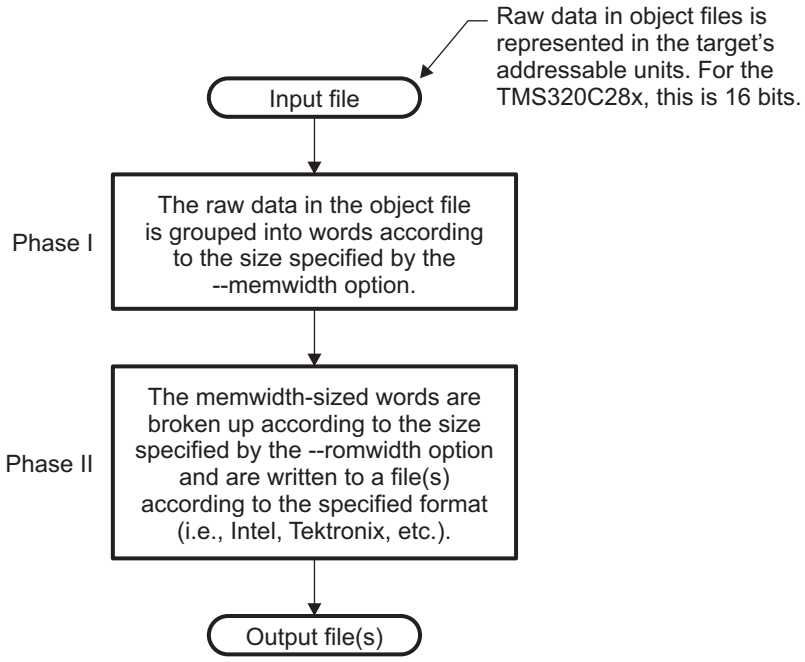
The hex conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex conversion utility, you must understand how the utility treats word widths. Three widths are important in the conversion process:

- Target width
- Memory width
- ROM width

The terms target word, memory word, and ROM word refer to a word of such a width.

Figure 12-2 illustrates the separate and distinct phases of the hex conversion utility's process flow.

Figure 12-2. Hex Conversion Utility Process Flow



12.3.1 Target Width

Target width is the unit size (in bits) of the target processor's word. The width is fixed for each target and cannot be changed. The TMS320C28x targets have a width of 16 bits.

12.3.2 Specifying the Memory Width

Memory width is the physical width (in bits) of the memory system. Usually, the memory system is physically the same width as the target processor width: a 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.

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 12.4.

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 when you need to break single target words into consecutive, narrower memory words.

Binary Format is 8 Bits Wide

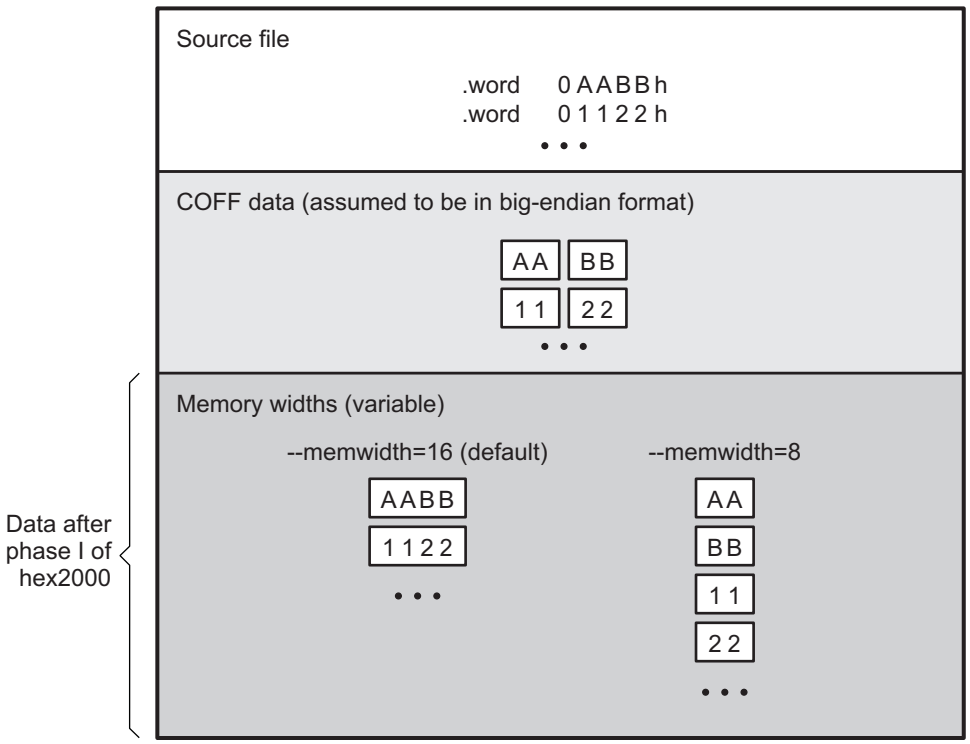
NOTE: You cannot change the memory width of the Binary format. The Binary hex format supports an 8-bit memory width only.

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 12-3 demonstrates how the memory width is related to object file data.

Figure 12-3. Object File Data and Memory Widths



12.3.3 Partitioning Data Into Output Files

If your *.out file contains sections allocated to multiple pages, separate output files are generated for each page. See Section 8.5.4.2 for information about specifying memory pages.

In addition, ROM width determines how the hex conversion utility partitions the 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). 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 12.4.

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, the utility simply writes multibyte fields into the file. The --romwidth option is ignored for the TI-TXT and TI-Tagged formats.

Figure 12-4 illustrates how the object file data, memory, and ROM widths are related to one another.

Memory width and ROM width are used only for grouping the object file data; they do not represent values. Thus, the byte ordering of the object file data is maintained throughout the conversion process. To refer to the partitions within a memory word, the bits of the memory word are always numbered from right to left as follows:

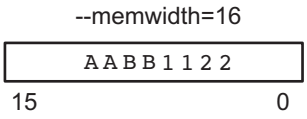
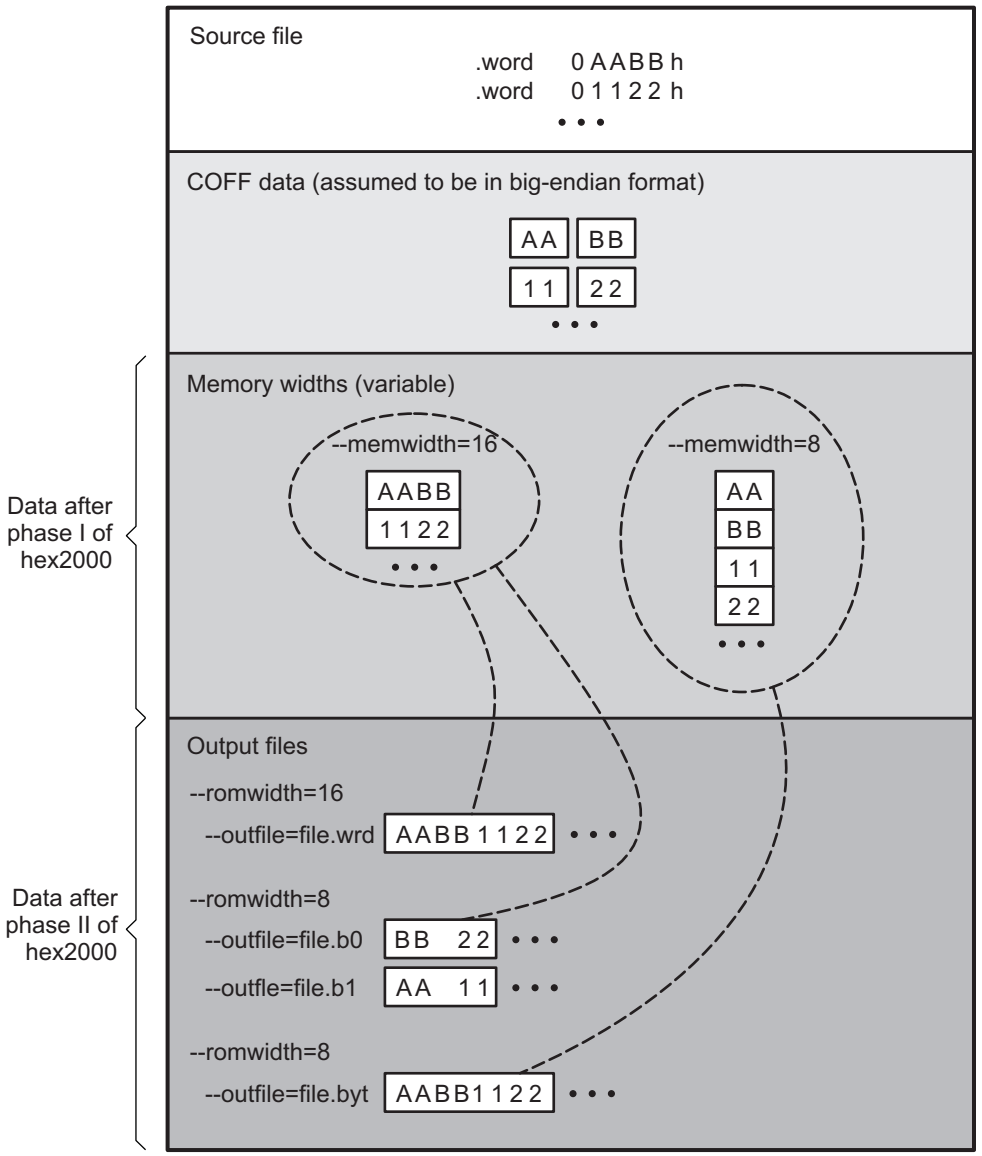


Figure 12-4. Data, Memory, and ROM Widths



12.3.4 Specifying Word Order for Output Words

There are two ways to split a wide word into consecutive memory locations in the same hex conversion utility output file:

- **--order=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 little-endian format. Unless your boot loader program expects big-endian format, avoid using --order=MS.

NOTE: **When the --order Option Applies**

- This option applies only when you use a memory width with a value less than 16. Otherwise, --order is ignored.
- This option does not affect the way memory words are split into output files. Think of the files as a set: the set contains a least significant file and a most significant file, but there is no ordering over the set. When you list filenames for a set of files, you always list the least significant first, regardless of the --order option.

12.4 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 TMS320C28x linker: both define the memory map of the target address space. Each line entry in the ROMS directive defines a specific address range. The general syntax is:

ROMS
{
romname **:** [**origin**=*value*,] [**length**=*value*,] [**romwidth**=*value*,]
[**memwidth**=*value*,] [**fill**=*value*]
[**files**=**{** *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.

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, it defaults to the length of the entire address space.

romwidth specifies the physical ROM width of the range in bits (see Section 12.3.3). 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 12.3.2). 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 12.5.)

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 12.9.2.)

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 12.3.3. 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.

12.4.1 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.

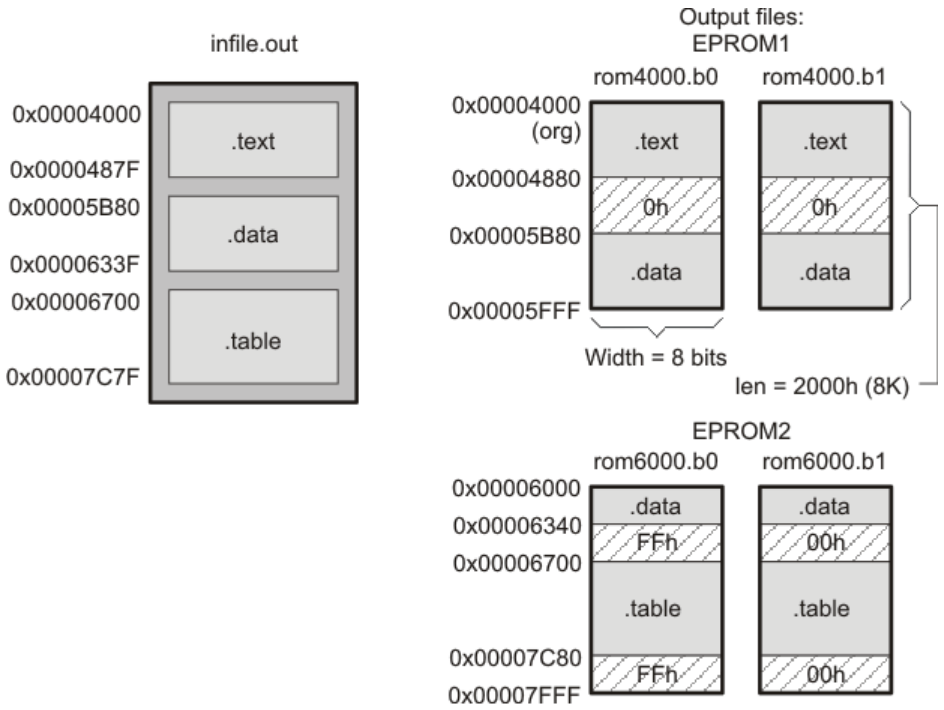
12.4.2 An Example of the ROMS Directive

The ROMS directive in Example 12-1 shows how 16K bytes of 16-bit memory could be partitioned for two 8K-byte 8-bit EPROMs. Figure 12-5 illustrates the input and output files.

Example 12-1. A ROMS Directive Example

```
infile.out
--image
--memwidth 16

ROMS
{
   EPROM1: org = 0x00004000, len = 0x2000, romwidth = 8
           files = { rom4000.b0, rom4000.b1}

   EPROM2: org = 0x00006000, len = 0x2000, romwidth = 8,
           fill = 0xFF00FF00,
           files = { rom6000.b0, rom6000.b1}
}
```

Figure 12-5. The infile.out File Partitioned Into Four Output Files



The map file (specified with the --map option) is advantageous when you use the ROMS directive with multiple ranges. The map file shows each range, its parameters, names of associated output files, and a list of contents (section names and fill values) broken down by address. Example 12-2 is a segment of the map file resulting from the example in Example 12-1.

Example 12-2. Map File Output From Example 12-1 Showing Memory Ranges

```
-------------------------------------------------------
00004000..00005fff Page=0 Width=8 "EPROM1"
-------------------------------------------------------
   OUTPUT FILES:   rom4000.b0   [b0..b7]
                   rom4000.b1   [b8..b15]
   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
-------------------------------------------------------
00006000..00007fff Page=0 Width=8 "EPROM2"
-------------------------------------------------------
   OUTPUT FILES:   rom6000.b0   [b0..b7]
                   rom6000.b1   [b8..b15]
   CONTENTS: 00006000..0000633f .data
             00006340..000066ff FILL = 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

12.5 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 TMS320C28x C/C++ compiler automatically generates these sections:

- **Initialized sections:** .text, .econst or .const (depending on the ABI), and .cinit
- **Uninitialized sections:** .ebss or .bss, .stack, and .esysmem or .sysmem

Use the SECTIONS directive in a command file. (See Section 12.2.2.) The general syntax 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, .econst, .vectors, .coeff, and .tables. If you want only .text and .data to be converted, use this a SECTIONS directive:

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

To configure both of these sections for boot loading, add the boot keyword:

```
SECTIONS { .text = boot .data = boot }
```

For more information about --boot and other command line options associated with boot tables, see Section 12.2 and Section 12.11.

12.6 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.

12.6.1 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.

12.6.2 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.

12.7 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, use 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, do not use quotes.

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.

12.8 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 your *.out file contains sections allocated to multiple pages, separate output files are generated for each page. See Section 8.5.4.2 for information about specifying memory pages.

   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 12.15):

      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

12.9 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.

12.9.1 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.

12.9.2 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=0x0FF results in a fill pattern of 0x0FF. 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.

12.9.3 Steps to Follow in Using Image Mode

Step 1: Define the ranges of target memory with a ROMS directive. See Section 12.4.

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.

12.10 Array Output Format

The --array option causes the output to be generated in C array format. In this format, data contained in initialized sections of an executable file are defined as C arrays. Output arrays may be compiled along with a host program and used to initialize the target at runtime.

Arrays are formed by collecting the initialized sections from the input executable. There are two ways arrays are formed:

- **With the ROMS directive.** Each memory range that is given in the ROMS directive denotes an array. The *romname* is used as the array name. The *origin* and *length* parameters of the ROM directive are required. The *memwidth*, *romwidth*, and *files* parameters are invalid and are ignored.
- **No ROMS directive (default).** If no ROMS directive is given, arrays are formed by combining initialized sections within each page, beginning with the first initialized section. Arrays will reflect any gaps that exist between sections.

  The default name for the array generated for `test.out` is test_image_0. The --array:name_prefix option can be used to override the default prefix for array names. For example, use --array:name_prefix=myarray to cause the name for the array to be myarray_0.

The data type for array elements is uint16_t.

12.11 Building a Table for an On-Chip Boot Loader

Some C28x devices, such as the F2810/12, 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 stored in memory or loaded from a device peripheral to initialize code or data. The hex conversion utility supports the boot loader by automatically building the boot table.

See Section 3.1.2 for a general discussion of bootstrap loading.

12.11.1 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. The table can be stored in memory (such as EPROM) or read in through a device peripheral (such as a serial or communications port).

The hex conversion utility automatically builds the boot table for the boot loader. Using the utility, you specify the sections you want the boot loader to initialize and the table location. The hex conversion utility builds a complete image of the table according to the format specified and converts it into hexadecimal in the output files. Then, you can burn the table into ROM or load it by other means.

The boot loader supports loading from memory that is narrower than the normal width of memory. For example, you can boot a 16-bit TMS320C28x from a single 8-bit EPROM by using the --memwidth option to configure the width of the boot table. The hex conversion utility automatically adjusts the table's format and length. See the boot loader example in the *TMS320C28x DSP CPU and Instruction Set Reference Guide* for an illustration of a boot table.

12.11.2 The Boot Table Format

The boot table format is simple. Typically, there is a header record containing a key value that indicates memory width, entry point, and values for 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. The table ends with a header containing size zero. See the boot loader section in the *TMS320C28x DSP CPU and Instruction Set Reference Guide* for more information.

12.11.3 How to Build the Boot Table

Table 12-2 summarizes the hex conversion utility options available for the boot loader.

Table 12-2. Boot-Loader Options

Option	Description
--boot	Convert all sections into bootable form (use instead of a SECTIONS directive).
--bootorg=*value*	Specify the source address of the boot-loader table.
--entrypoint=*value*	Specify the entry point at which to begin execution after boot loading. The *value* can be an address or a global symbol.
--gpio8	Specify the source of the boot-loader table as the GP I/O port, 8-bit mode
--gpio16	Specify the source of the boot-loader table as the GP I/O port, 16-bit mode
--lospcp=*value*	Specify the initial value for the LOSPCP register. The value is used only for the spi8 boot table format and is ignored for all other formats. A value greater than 0x7F is truncated to 0x7F.
--sci8	Specify the source of the boot-loader table as the SCI-A port, 8-bit mode
--spi8	Specify the source of the boot-loader table as the SPI-A port, 8-bit mode
--spibrr=*value*	Specify the initial value for the SPIBRR register. The value is used only for the spi8 boot table format and is ignored for all other formats. A value greater than 0x7F is truncated to 0x7F.

12.11.3.1 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 12.5).

When you select a section for placement in a boot-loader table, the hex conversion utility places the section's *load address* in the destination address field for the block in the boot table. The section content is then treated as raw data for that block. *The hex conversion utility does not use the section run address.* When linking, you need not worry about the ROM address or the construction of the boot table; the hex conversion utility handles this.

Step 2: **Identify the bootable sections.** You can use the --boot option to tell the hex conversion utility to configure all sections for boot loading. Or, you can use a SECTIONS directive to select specific sections to be configured (see Section 12.5). If you use a SECTIONS directive, the --boot option is ignored.

Step 3: **Set the boot table format.** Specify the --gpio8, --gpio16, --sci8, or --spi8 options to set the source format of the boot table. You do not need to specify the memwidth and romwidth as the utility will set these formats automatically. If --memwidth and --romwidth are used after a format option, they override the default for the format.

Step 4: **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 C28x and booting from memory location 0x3FF000, specify --bootorg=0x3FF000. The address field for the boot table in the hex conversion utility output file will then start at 0x3FF000.

Step 5: **Set boot-loader-specific options.** Set entry point and control register values as needed.

Step 6: **Describe your system memory configuration.** See Section 12.3 and Section 12.4.

12.11.3.2 Leaving Room for the Boot Table

The complete boot table is similar to a single section containing all of the header records and data for the boot loader. The address of this section is the boot table origin. As part of the normal conversion process, the hex conversion utility converts the boot table to hexadecimal format and maps it into the output files like any other section.

Be sure to leave room in your system memory for the boot table, especially when you are using the ROMS directive. The boot table cannot overlap other nonboot sections or unconfigured memory. Usually, this is not a problem; typically, a portion of memory in your system is reserved for the boot table. Simply configure this memory as one or more ranges in the ROMS directive, and use the --bootorg option to specify the starting address.

12.11.4 Booting From a Device Peripheral

You can choose to boot from the F2810/12 serial or parallel port by using the --gpio9, --gpio16, --sci8, or --spi8 boot table format option. The initial value for the LOSPCP register can be specified with the --lospcp option. The initial value for the SPIBRR register can be specified with the --spibrr option. Only the --spi8 format uses these control register values in the boot table.

If the register values are not specified for the --spi8 format, the hex conversion utility uses the default values 0x02 for LOSPCP and 0x7F for SPIBRR. When the boot table format options are specified and the ROMS directive is not specified, the ASCII format hex utility output does not produce the address record.

12.11.5 Setting the Entry Point for the Boot Table

After completing the boot load process, execution starts at the default entry point specified by the linker and contained in the object file. By using the --entrypoint 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 --entrypoint=0x0123 on the command line or in a command file. You can determine the --entrypoint 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.

12.11.6 Using the C28x Boot Loader

This subsection explains how to use the hex conversion utility with the boot loader for C28x devices. The C28x boot loader accepts the formats listed in Table 12-3.

Table 12-3. Boot Table Source Formats

Format	Option
Parallel boot GP I/O 8 bit	--gpio8
Parallel boot GP I/O 16 bit	--gpio16
8-bit SCI boot	--sci8
8-bit SPI boot	--spi8

The F2810/12 can boot through the SCI-A 8-bit, SPI-A 8-bit, GP I/O 8-bit, or GP I/I 16-bit interface. The format of the boot table is shown in Table 12-4.

Table 12-4. Boot Table Format

Description	Word	Content
Boot table header	1	Key value (0x10AA or 0x08AA)
	2-9	Register initialization value or reserved for future use
	10-11	Entry point
Block header	12	Block size in number of words (n1)
	13-14	Destination address of the block
Block data	15	Raw data for the block (n1 words)
Block header	16 + nl	Block size in number of words
	.	Destination address of the block
Block data	.	Raw data for the block
Additional block headers and data, as required	...	Content as appropriate
Block header with size 0		0x0000; indicates the end of the boot table.

The C28x can boot through either the serial 8-bit or parallel interface with either 8- or 16-bit data. The format is the same for any combination: the boot table consists of a field containing the destination address, a field containing the length, and a block containing the data. You can boot only one section. If you are booting from an 8-bit channel, 16-bit words are stored in the table with MSBs first; the hex conversion utility automatically builds the table in the correct format. Use the following options to specify the boot table source:

- To boot from a SCI-A port, specify --sci8 when invoking the utility. Do not specify --memwidth or --romwidth.
- To boot from a SPI-A port, specify --spi8 when invoking the utility. Do not specify --memwidth or --romwidth. Use --lospcp to set the initial value for the LOSPCP register and --spibrr to set the initial value for the SPIBRR register. If the register values are not specified for the --spi8 format, the hex conversion utility uses the default value 0x02 for LOSPCP and 0x7F for SPIBRR.
- To load from a general-purpose parallel I/O port, invoke the utility with --gpio8 or --gpio16. Do not specify --memwidth or --romwidth.

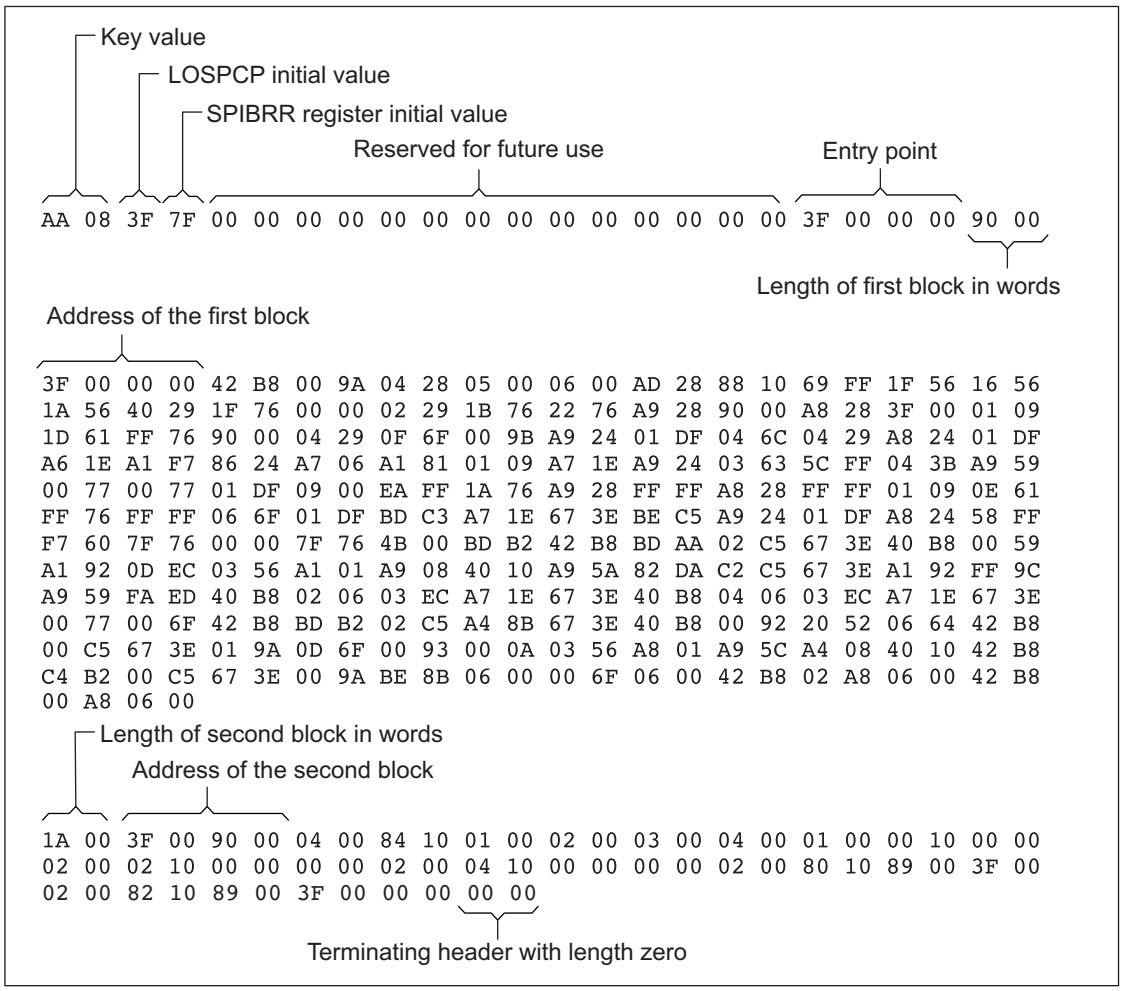
The command file in Example 12-3 allows you to boot the .text and .cinit sections of test.out from a 16-bit-wide EPROM at location 0x3FFC00. The map file test.map is also generated.

Example 12-3. Sample Command File for Booting From 8-Bit SPI Boot

```
/*--------------------------------------------------------------------------*/
/* Hex converter command file.                                              */
/*--------------------------------------------------------------------------*/
test.out                /* Input file */
--ascii                 /* Select ASCII format */
--map=test.map          /* Specify the map file */
--outfile=test_spi8.hex /* Hex utility out file */
--boot                  /* Consider all the input sections as boot sections */
--spi8                  /* Specify the SPI 8-bit boot format */
--lospcp=0x3F           /* Set the initial value for the LOSPCP as 0x3F */
                        /* The -spibrr option is not specified to show that */
                        /* the hex utility uses the default value (0x7F) */
--entrypoint=0x3F0000   /* Set the entry point */
```

The command file in Example 12-3 generates the out file in Figure 12-6. The control register values are coded in the boot table header and that header has the address that is specified with the --entrypoint option.

Figure 12-6. Sample Hex Converter Out File for Booting From 8-Bit SPI Boot



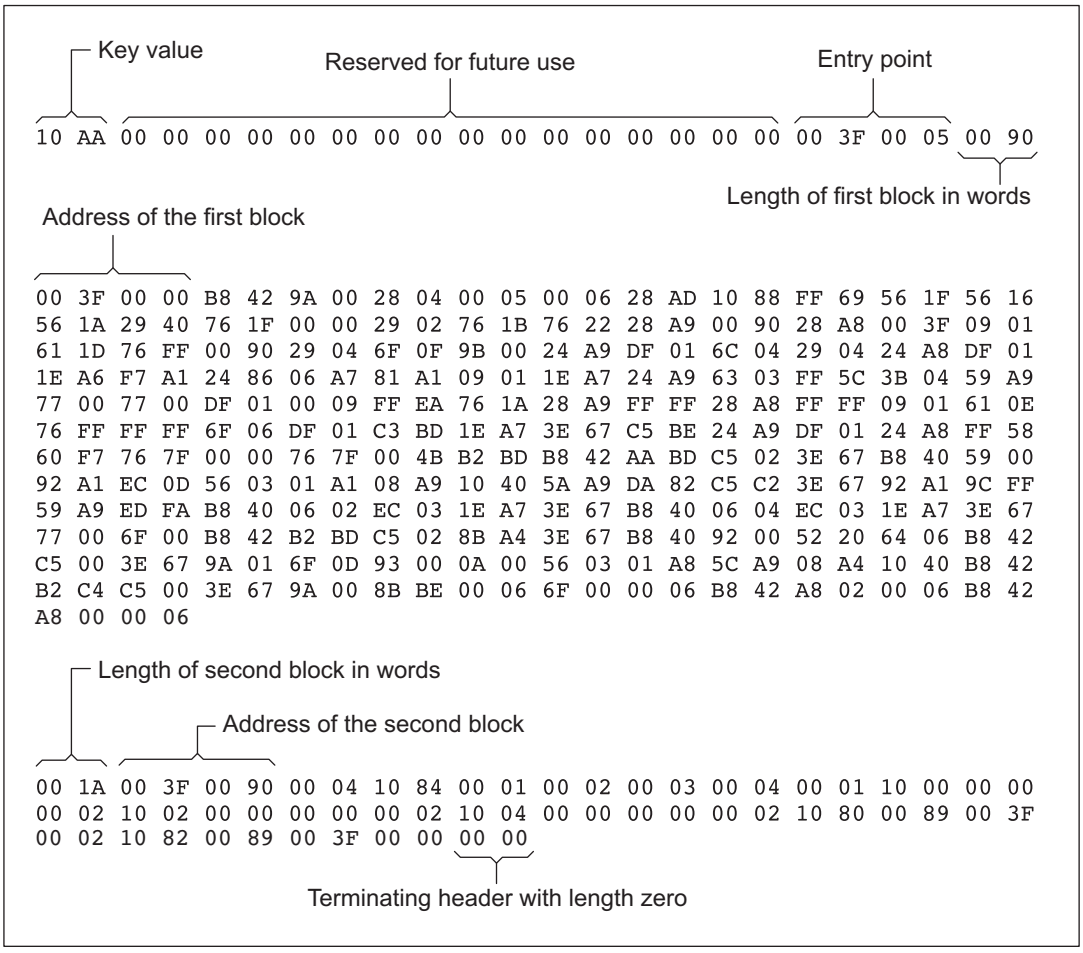
The command file in Example 12-4 allows you to boot the .text and .cinit sections of test.out from the 16-bit parallel GP I/O port. The map file test.map is also generated.

Example 12-4. Sample Command File for C28x 16-Bit Parallel Boot GP I/O

```
/*--------------------------------------------------------------------*/
/* Hex converter command file.                                        */
/*--------------------------------------------------------------------*/
test.out                  /* Input file */
--ascii                   /* Select ASCII format */
--map=test.map            /* Specify the map file */
--outfile=test_gpio16.hex  /* Hex utility out file */
--gpio16                  /* Specify the 16-bit GP I/O boot format */

SECTIONS
{
   .text: paddr=BOOT
   .cinit: paddr=BOOT
}
```

The command file in Example 12-4 generates the out file in Figure 12-7.

Figure 12-7. Sample Hex Converter Out File for C28x 16-Bit Parallel Boot GP I/O



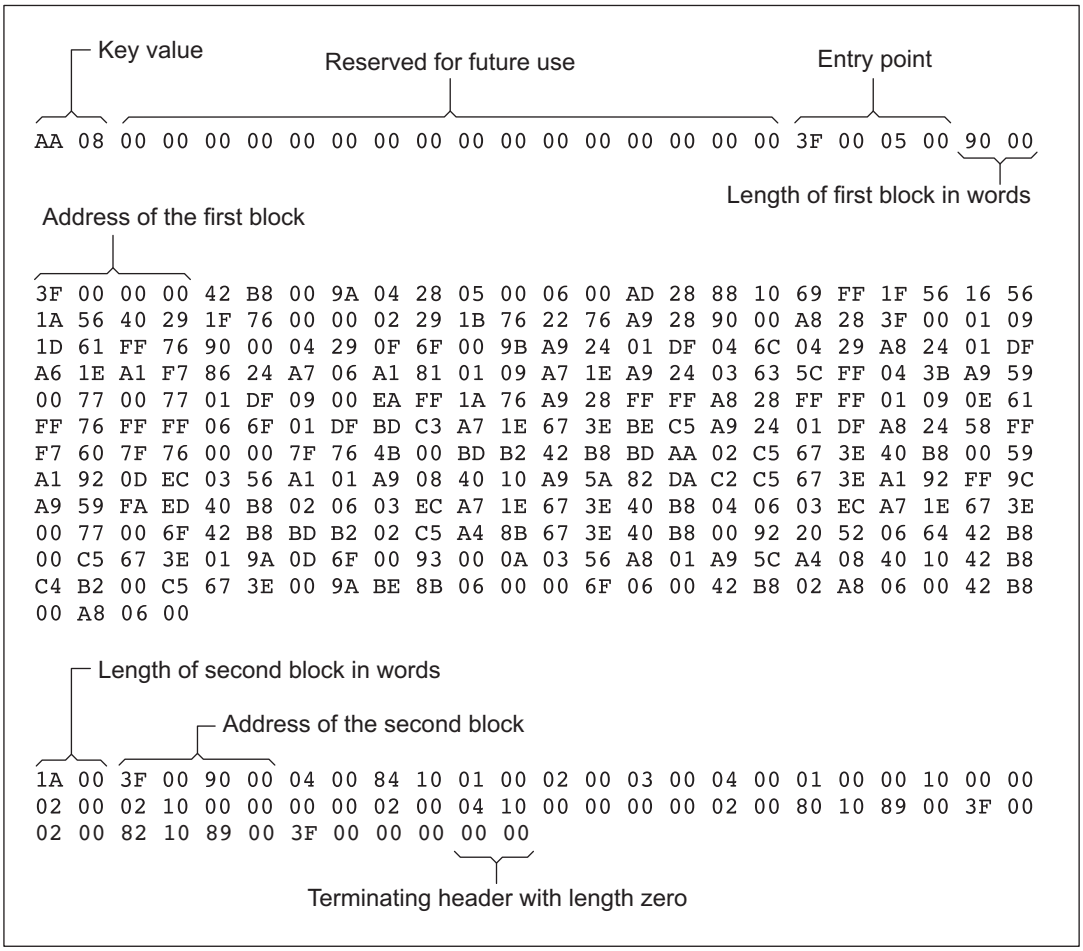
The command file in Example 12-5 allows you to boot the .text and .cinit sections of test.out from a 16-bit wide EPROM from the SCI-A 8-bit port. The map file test.map is also generated.

Example 12-5. Sample Command File for Booting From 8-Bit SCI Boot

```
/*-----------------------------------------------------------------------*/
/* Hex converter command file.                                           */
/*-----------------------------------------------------------------------*/
test.out                  /* Input file */
-ascii                    /* Select ASCII format */
--map=test.map            /* Specify the map file */
--outfile=test_sci8.hex   /* Hex utility out file */
--sci8                    /* Specify the SCI 8-bit boot format */

SECTIONS
{
   .text: paddr=BOOT
   .cinit: paddr=BOOT
}
```

The command file in Example 12-5 generates the out file in Figure 12-8.

Figure 12-8. Sample Hex Converter Out File for Booting From 8-Bit SCI Boot



12.12 Using Secure Flash Boot on TMS320F2838x Devices

The hex conversion utility supports the secure flash boot capability provided by TMS320F2838x devices, which have both C28 and ARM cores. The secure flash boot applies the Cipher-based Message Authentication Protocol (CMAC) algorithm to verify CMAC tags for regions of allocated memory.

Secure flash boot is similar to the regular flash boot mode in that the boot flow branches to the configured memory address in flash. The difference is that this branch occurs only after the flash memory contents have been authenticated. The flash authentication uses CMAC to authenticate 16 KB of flash. The CMAC calculation requires a 128-bit key that you define. Additionally, you must calculate a golden CMAC tag based on the 16 KB flash memory range and store it along with the application code at a hardcoded address in flash. During secure flash boot, the calculated CMAC tag is compared to the golden CMAC tag in flash to determine the pass/fail status of the CMAC authentication. If authentication passes, the boot flow continues and branches to flash to begin executing the application. See the *TMS320F2838x Microcontrollers Technical Reference Manual* (SPRUII0) for further details about secure flash boot and the CMAC algorithm.

In order to apply the CMAC algorithm to the appropriate regions in allocated memory, use the hex conversion utility as follows:

- Use the --cmac=file option. The file should contain a 128-bit hex CMAC key.

  The CMAC key in the file specified by the --cmac command-line option must use the format 0xkey0key1key2key3 in order to access the device registers for CMACKEY0-3. For example, the following file contents represent CMACKEY registers containing key0=`0x7c0b7db9`, key1=`0x811f10d0`, key2=`0x0e476c7a`, and key3=`0x0d92f6e0`.

  ```
  0x7c0b7db9811f10d00e476c7a0d92f6e0
  ```

- Use either the --image option or the --load_image option when using the --cmac option.
  - If you use the --image option, set both --memwidth and --romwidth to 16.
- Do not use the --boot option with the --cmac option.
- Specify a HEX directive with one entry that represents all the allocated flash memory. Use a 128-bit aligned length and specify the optional fill value. (The default fill is set to 0's.)
- Define the global CMAC tags in C code.

The CMAC feature uses four secure flash boot memory regions that are hardcoded for start/end/tag addresses, and one flexible CMAC region. The flexible region can encompass the entire allocated region as input in the HEX directive or user-specified start/end addresses defined in C code.

C code definitions like the following are required to reserve space for the CMAC tag symbols.

```
struct CMAC_TAG
{  char     tag[8];
   uint32_t start;
   uint32_t end;
};

#pragma RETAIN(cmac_sb_1)
#pragma LOCATION(cmac_sb_1, 0x080002)
const char cmac_sb_1[8] = { 0 };

#pragma RETAIN(cmac_sb_2)
#pragma LOCATION(cmac_sb_2, 0x088002)
const char cmac_sb_2[8] = { 0 };

#pragma RETAIN(cmac_sb_3)
#pragma LOCATION(cmac_sb_3, 0x0a8002)
const char cmac_sb_3[8] = { 0 };

#pragma RETAIN(cmac_sb_4)
#pragma LOCATION(cmac_sb_4, 0x0be002)
const char cmac_sb_4[8] = { 0 };

#pragma RETAIN(cmac_all)
#pragma LOCATION(cmac_all, 0x087002)
const struct CMAC_TAG cmac_all = { 0 }, 0x0, 0x0};
```

The four secure flash boot region CMAC tags are stored in the `cmac_sb_1` through `cmac_sb_4` symbols. The `cmac_all` symbol stores the CMAC tag for the flexible user-specified region. For `cmac_all`:

- If the `start` and `end` CMAC_TAG struct members are zero, then the CMAC algorithm runs over entire memory region specified in the HEX directive. The hex conversion utility populates the start and end memory locations with the addresses input from the HEX directive entry.
- If the `start` and `end` members are non-zero, then the CMAC algorithm is instead applied between the specified addresses.

RETAIN pragmas are required in the C code if these symbols are not accessed in the application code.

LOCATION pragmas are required to place symbols at the required memory locations. The LOCATION entries for `cmac_sb_1` through `cmac_sb_4` are at fixed addresses. The LOCATION address for `cmac_all` can be user-specified. However, it must not be located within any secure flash boot regions, because the ROM CMAC implementation on the devices does not support this.

The CMAC algorithm is applied prior to the hex conversion. No changes are made to the original input ELF executable.

The hex conversion utility applies the CMAC algorithm only to CMAC regions that have global symbols defined. So if an ELF executable defines only `cmac_sb_1` and `cmac_all`, then only those two CMAC tags will be generated and populated in the generated hex output file.

12.13 Controlling the ROM Device Address

The hex conversion utility output address field corresponds to the ROM device address. The EPROM programmer burns the data into the location specified by the hex conversion utility output file address field. The hex conversion utility offers some mechanisms to control the starting address in ROM of each section. However, many EPROM programmers offer direct control of the location in ROM in which the data is burned.

The address field of the hex-conversion utility output file is controlled by the following items, which are listed from low to high priority:

1. **The linker command file.** By default, the address field of the hex conversion utility output file is the load address (as given in the linker command file).
2. **The paddr parameter of the SECTIONS directive.** When the paddr parameter is specified for a section, the hex conversion utility bypasses the section load address and places the section in the address specified by paddr.
3. **The --zero option.** When you use the --zero option, the utility resets the address origin to 0 for each output file. Since each file starts at 0 and counts upward, any address records represent offsets from the beginning of the file (the address within the ROM) rather than actual target addresses of the data.

   You must use the --zero option in conjunction with the --image option to force the starting address in each output file to be zero. If you specify the --zero option without the --image option, the utility issues a warning and ignores the --zero option.
4. **The --byte option.** Some EPROM programmers may require the output file address field to contain a byte count rather than a word count. If you use the −byte option, the output file address increments once for each byte. For example, if the starting address is 0h, the first line contains eight words, and you use no −byte option, the second line would start at address 8 (8h). If the starting address is 0h, the first line contains eight words, and you use the −byte option, the second line would start at address 16 (010h). The data in both examples are the same; −byte affects only the calculation of the output file address field, not the actual target processor address of the converted data.

   The --byte option causes the address records in an output file to refer to byte locations within the file, whether the target processor is byte-addressable or not.

12.14 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 *TMS320C28x 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

12.15 Description of the Object Formats

The hex conversion utility has options that identify each format. Table 12-5 specifies the format options. They are described in the following sections.

- You should 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 12-5. 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

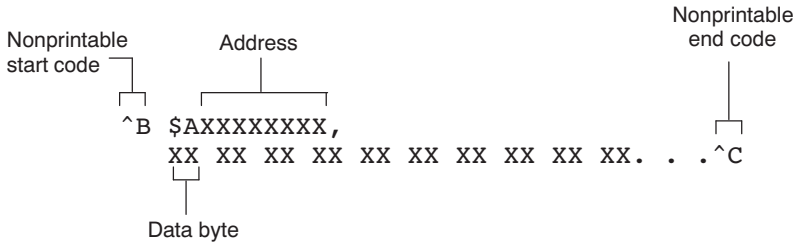
Address bits determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

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.

12.15.1 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 12-9 illustrates the ASCII-Hex format.

Figure 12-9. 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.

12.15.2 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

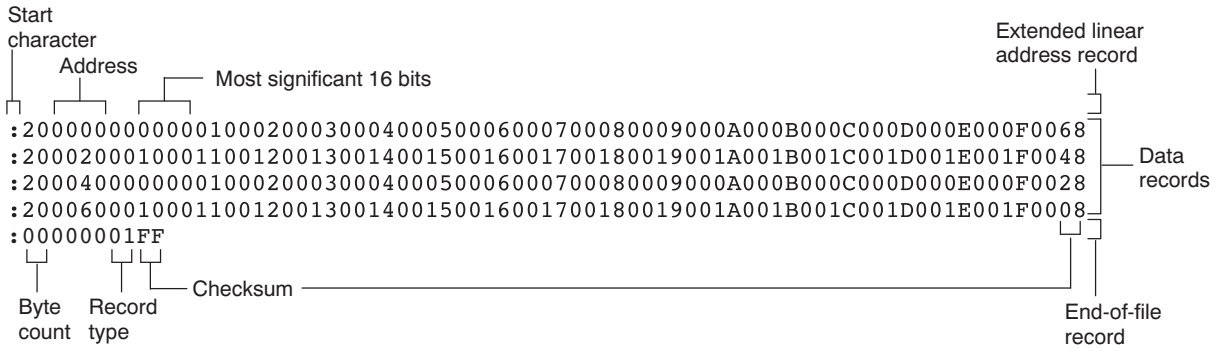
Record type00, the data record, begins with a colon (:) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended 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 12-10 illustrates the Intel hexadecimal object format.

Figure 12-10. Intel Hexadecimal Object Format



12.15.3 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:

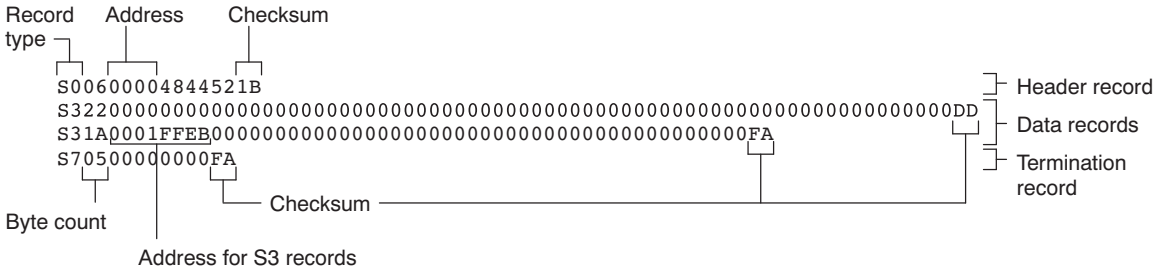
Record Type	Description
S0	Header record
S1	Code/data record for 16-bit addresses (S1 format)
S2	Code/data record for 24-bit addresses (S2 format)
S3	Code/data record for 32-bit addresses (S3 format)
S7	Termination record for 32-bit addresses (S3 format)
S8	Termination record for 24-bit addresses (S2 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 12-11 illustrates the Motorola-S object format.

Figure 12-11. Motorola-S Format



12.15.4 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.

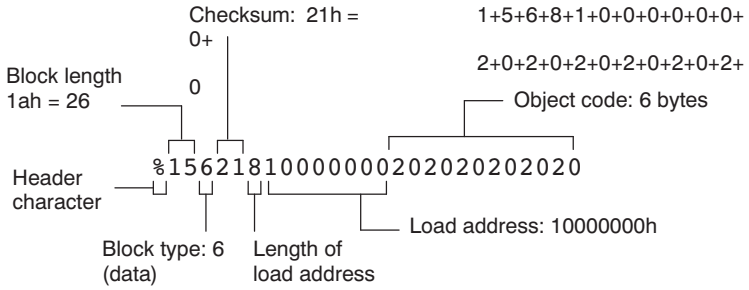
The header field in the data record contains the following information:

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 12-12 illustrates the Tektronix object format.

Figure 12-12. Extended Tektronix Object Format



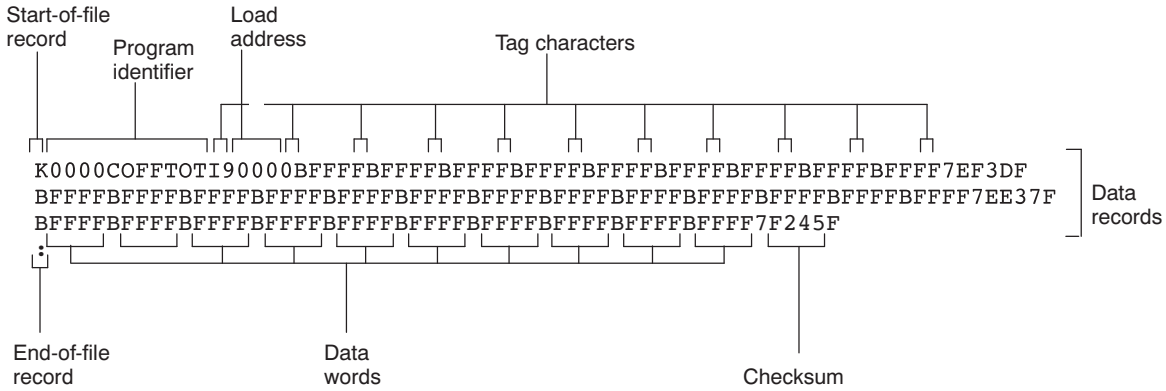
12.15.5 Texas Instruments SDSMAC (TI-Tagged) Object Format (--ti_tagged Option)

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 12-13 illustrates the tag characters and fields in TI-Tagged object format.

Figure 12-13. TI-Tagged Object Format



If any data fields appear before the first address, the first field is assigned address 0000h. Address fields may be expressed 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 (:).

12.15.6 TI-TXT Hex Format (--ti_txt Option)

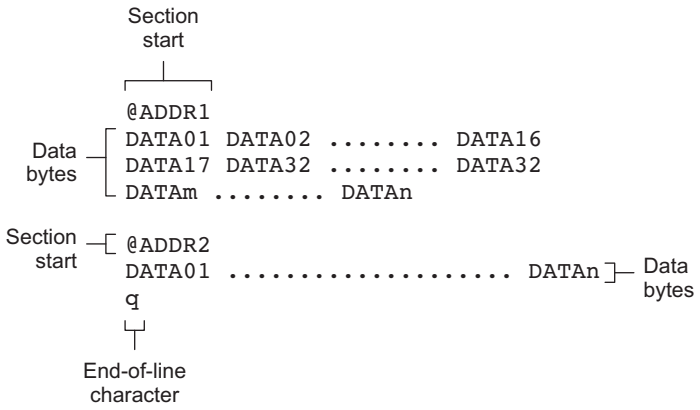
The TI-TXT hex format supports 16-bit hexadecimal data. It consists of section start addresses, data byte, and an end-of-file character. These restrictions apply:

- 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 12-14. TI-TXT Object Format



Example 12-6. 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
```

12.16 Hex Conversion Utility Error Messages

section mapped to reserved memory

Description A section is mapped into a memory area that is designated as reserved in the processor memory map.

Action Correct section or boot-loader address. For valid memory locations, refer to the *TMS320C28x CPU and Instruction Set Reference Guide.*

sections overlapping

Description Two or more 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 hexadecimal output-file address that is performed by the hex-conversion utility when memory width is less than data width. See Section 12.3 and Section 12.13.

unconfigured memory error

Description The file contains a section whose load address falls outside the memory range defined in the ROMS directive.

Action Correct the ROM range as defined by the ROMS directive to cover the memory range needed, or modify the section load address. Remember that if the ROMS directive is not used, the memory range defaults to the entire processor address space. For this reason, removing the ROMS directive could also be a workaround.

Sharing C/C++ Header Files With Assembly Source

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code.

Topic **Page**

13.1 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.

13.2 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.

13.2.1 Comments

Comments are consumed entirely at the C level, and do not appear in the resulting converted assembly file.

13.2.2 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.

13.2.3 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.

13.2.4 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.

13.2.5 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.

13.2.6 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.

13.2.7 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.

13.2.8 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 13.2.11 for suggestions on how to use C/C++ macro strings.

Macros are converted using the .define directive (see Section 13.4.2), 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 13.4.3).

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.

13.2.9 The #undef Directive

Symbols undefined using the #undef directive before the end of the .cdecls are not converted to assembly.

13.2.10 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.

13.2.11 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 13.4.7 for the .cstring directive syntax.)

13.2.12 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.

13.2.13 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 13.2.3 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.

13.2.14 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 13.3.1 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 13.2.13 for information on variables names which are of a structure/union type.

13.2.15 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.

13.2.16 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.

13.3 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.

13.3.1 Name Mangling

C++ compilers use *name mangling* to avoid conflicts between identically named functions and variables. If name mangling were not used, symbol name clashes can occur.

You can use the demangler (dem2000) to demangle names and identify the correct symbols to use in assembly. See the "C++ Name Demangler" chapter of the *TMS320C28x Optimizing C/C++ Compiler User's Guide* for details.

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);
      ...
   }
```

13.3.2 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.

13.3.3 Templates

No support exists for templates.

13.3.4 Virtual Functions

No support exists for virtual functions, as they have no assembly representation.

13.4 Special Assembler Support

13.4.1 Enumerations (.enum/.emember/.endenum)

The following directives support a pseudo-scoping for enumerations:

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.

13.4.2 The .define Directive

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 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.

13.4.3 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 13.4.2, which covers the .define directive.

13.4.4 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 13.4.2 and Section 13.4.3 for the use of .define and .undef in assembly.

13.4.5 The $sizeof Built-In Function

The 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 13.2.12, which notes that this conversion does not happen automatically if the C/C++ sizeof( ) built-in function is used within a macro.

13.4.6 Structure/Union Alignment and $alignof( )

The assembly .struct and .union directives 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.

13.4.7 The .cstring Directive

You can use the .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 13.2.11 for more information on the .cstring directive.

Symbolic Debugging Directives

The assembler supports several directives that the TMS320C28x 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:

```
cl2000 --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:

```
cl2000 --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 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 *TMS320C28x Optimizing C/C++ Compiler User's Guide.*

Table A-1. Symbolic Debugging Directives

Label	Directive	Arguments
	.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* **)** [**,** ...]

XML Link Information File Description

The TMS320C28x 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>.ebss</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 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-6. 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>
```

CRC Reference Implementation

This appendix contains source code in C for a reference implementation of a CRC calculation routine that is compatible with the linker-generated CRC tables. This code is found in the file labeled ref_crc.c.

This appendix also contains source code for a simple example application using linker-generated CRC tables and copy tables. The application contains several tasks which share a common run area. Linker-generated copy tables move the tasks from their load addresses to the run address. The application also uses the reference CRC calculation routine to compute CRC values which are compared against the linker-generated values.

This code is for reference only, and no warranty is made as to suitability for any purpose.

Topic **Page**

C.1 Compilation Instructions

1. Run a stand-alone test of the reference implementation of CRC computation.
   - For C2000 on Linux:

     ```
     cl2000 -D=_RUN_MAIN ref_crc.c -z -o ref_crc_c2000 -l lnk2800_ml.cmd
     <Linking>
     ```

     Run ref_crc_c2000 with an appropriate simulator:
     - CRC-32-PRIME: 4beab53b
     - CRC-8-PRIME: 70
     - CRC16_802_15_4: 1bd3
   - For GCC on Linux:

     ```
     gcc -D_RUN_MAIN -o ref_crc_gcc ref_crc.c
     ./ref_crc_gcc
     ```

     Run ref_crc_gcc with an appropriate simulator:
     - CRC-32-PRIME: 4beab53b
     - CRC-8-PRIME: 70
     - CRC16_802_15_4: 1bd3
   -
2. Run a simple example program using copy tables and CRC tables.
   - For C2000 on Linux:

     ```
     cl2000 -c *.c
     cl2000 -z -lex1.cmd
     <Linking>
     ```

     Run ex1.out with an appropriate simulator:
     - CRC-32-PRIME: 4beab53b
     - CRC-8-PRIME: 70
     - CRC16_802_15_4: 1bd3

C.2 Reference CRC Calculation Routine

Example C-1. Reference Implementation of a CRC Calculation Function: ref_crc.c

```
/*****************************************************************************/
/* Reference implementation of a CRC calculation function                    */
/*                                                                           */
/* gen_crc is the interface function which should be called from the         */
/* application.  There is also a stand-alone test mode that can be used      */
/* if _RUN_MAIN is defined.                                                  */
/*****************************************************************************/

/*---------------------------------------------------------------------------*/
/* This file does NOT implement a general-purpose CRC function.             */
/* Specifically, it does not handle parameterization by initial value, bit   */
/* reflection, or final XOR value.  This implementation is intended only to  */
/* implement the CRC functions used by the linker for C28x CRC tables.  The  */
/* algorithms used by the linker are selected to match the CRC algorithms in */
/* the PRIME and IEEE 802.15.4-2006 standards, which use the polynomials     */
/* supported by the C28x VCU hardware.  To understand CRCs in general,       */
/* especially what other parameters exist, see:                              */
/*                                                                           */
/* "A Painless Guide To CRC Error Detection Algorithms" likely at:           */
/*         http://www.ross.net/crc/download/crc_v3.txt                       */
/* Author : Ross Williams (ross@guest.adelaide.edu.au.).                     */
/* Date   : 3 June 1993.                                                     */
/* Status : Public domain (C code).                                          */
/*---------------------------------------------------------------------------*/
```

Example C-1. Reference Implementation of a CRC Calculation Function: ref_crc.c (continued)

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/*---------------------------------------------------------------------------*/
/* These are the CRC algorithms supported by the linker, which match the     */
/* polynomials supported in C28x VCU hardware, which match the PRIME and     */
/* IEEE 802.15.4-2006 standards.  These must match the values in crc_tbl.h.  */
/*---------------------------------------------------------------------------*/
#define CRC32_PRIME        0
#define CRC16_802_15_4     1
#define CRC16_ALT          2
#define CRC8_PRIME         3


typedef struct crc_config_t
{
    int id;
    int degree;
    unsigned long poly;
} crc_config_t;


const crc_config_t crc_config[] = { { CRC32_PRIME,    32, 0x04c11db7 },
                                    { CRC16_802_15_4, 16,     0x1021 },
                                    { CRC16_ALT,      16,     0x8005 },
                                    { CRC8_PRIME,      8,       0x07 } };

unsigned long crc_table[256] = { 0 };

const crc_config_t *find_config(int id) {
    size_t i;
    for (i = 0; i < sizeof(crc_config) / sizeof(*crc_config); i++)
        if (crc_config[i].id == id)
            return &crc_config[i];

    fprintf(stderr, "invalid config id %d\n", id);
    exit(EXIT_FAILURE);
    return NULL;
}

/*---------------------------------------------------------------------------*/
/* Table-driven version                                                      */
/*---------------------------------------------------------------------------*/
unsigned long generate_mask(int degree)
{
    unsigned long half = (1ul << (degree / 2)) - 1;
    return half << (degree / 2) | half;
}

void generate_crc_table(const crc_config_t *config)
{
    int i, j;

    unsigned long bit, crc;
    unsigned long high_bit = (1ul << (config->degree - 1));
    unsigned long mask     = generate_mask(config->degree);

    for (i = 0; i < 256; i++)
    {
        crc = (unsigned long)i << config->degree - 8;

        for (j = 0; j < 8; j++)
```

Example C-1. Reference Implementation of a CRC Calculation Function: ref_crc.c (continued)

```
        {
            bit = crc & high_bit;
            crc <<= 1;
            if (bit) crc^= config->poly;
        }

        crc_table[i] = crc & mask;
    }
}

/*****************************************************************************/
/* gen_crc - Return the CRC value for the data using the given CRC algorithm */
/*           int id         : identifies the CRC algorithm                   */
/*           char *data     : the data                                       */
/*           size_t len     : the size of the data                           */
/*****************************************************************************/
unsigned long gen_crc(int id, const unsigned char *data, size_t len)

{
    /*-------------------------------------------------------------------------*/
    /* Note: this is not a general-purpose CRC function.  It does not handle */
    /* parameterization by initial value, bit reflection, or final XOR       */
    /* value.  This CRC function is specialized to the CRC algorithms in the */
    /* linker used for C28x CRC tables.                                       */
    /*-------------------------------------------------------------------------*/

    /*-------------------------------------------------------------------------*/
    /* This CRC function is not intended to be optimal; it is written such   */
    /* that it works and generates the same result on all 8-bit and 16-bit   */
    /* targets, including C28x, other TI DSPs, and typical desktops.         */
    /*-------------------------------------------------------------------------*/

    const crc_config_t *config = find_config(id);

    unsigned long crc = 0;
    unsigned long mask = generate_mask(config->degree);

    size_t i;

    generate_crc_table(config);

    for (i = 0; i < len; i++)
    {
       unsigned int datum = data[i];

       /*---------------------------------------------------------------------*/
       /* This loop handles 16-bit chars when we compile on 16-bit machines. */
       /*---------------------------------------------------------------------*/
       int n;

       for (n = 0; n < (CHAR_BIT / 8); n++)
       {
           /*-----------------------------------------------------------------*/
           /* For 16-bit machines, we need to feed the octets in an          */
           /* arbitrary order.  For C2000, the arbitrary order we choose is  */
           /* to feed the LEAST significant octet of char 0 first.  The      */
           /* first octet fed to the CRC is the LEAST-significant octet of   */
           /* char 0; the second octet is the MOST-significant octet of char */
           /* 0.  See the "Special Note regarding 16-bit char" in the        */
           /* Assembly Language Tools User's Guide.                          */
           /*-----------------------------------------------------------------*/

#if __TMS320C28XX__
           /*-----------------------------------------------------------------*/
```

Example C-1. Reference Implementation of a CRC Calculation Function: ref_crc.c (continued)

```
            /* Using __byte is not necessary; we use it here to illustrate    */
            /* how it relates to octet order.                                 */
            /*----------------------------------------------------------------*/
            unsigned long octet = __byte((int*)&datum, n);
#else
            unsigned long octet = ((datum >> (8 * n)) & 0xff);
#endif
            unsigned long term1 = (crc << 8);
            int           idx   = ((crc >> (config->degree - 8)) & 0xff) ^ octet;

            crc = term1 ^ crc_table[idx];
        }
    }

    return crc & mask;
}

#ifdef _RUN_MAIN
/*****************************************************************************/
/* main - If requested, compute the CRC of test data using each algorithm.   */
/*****************************************************************************/
int main(void)
{
#if CHAR_BIT == 16
    const unsigned char data[] = { 'a', 'b', 'c', 'd' };
#elif CHAR_BIT == 8
    /*--------------------------------------------------------------------*/
    /* This represents "abcd" as it would appear in C2000 memory if we view  */
    /* C2000 memory as octets, least-significant octet first; see "a special */
    /* note regarding 16-bit char" in Assembly Language Tools User's Guide.  */
    /*--------------------------------------------------------------------*/
    const unsigned char data[] = { 'a', 0, 'b', 0, 'c', 0, 'd', 0 };
#endif
    /* CRC_8_PRIME: 0x70 */
    /* CRC_16_802: 0x1bd3 */
    /* CRC_32_PRIME: 0x4beab53b */

   const unsigned char *p = (const unsigned char *)data;

   unsigned long crc;

   crc = gen_crc(CRC32_PRIME, p, sizeof data);
   printf("CRC_32_PRIME: %08lx\n", crc);

   crc = gen_crc(CRC8_PRIME, p, sizeof data);
   printf("CRC_8_PRIME: %02lx\n", crc);

   crc = gen_crc(CRC16_802_15_4, p, sizeof data);
   printf("CRC16_802_15_4: %04lx\n", crc);

   return 0;
}
#endif
```

C.3 Linker-Generated Copy Tables and CRC Tables

Three tasks exist in separate load areas. As each is needed, it is copied into the common run area and executed. A separate copy table is generated for each task (see table() operator in ex1.cmd). CRC values for the task functions are verified as well. See Example C-7 for the crc_table() operator calls.

Example C-2. Main Routine for Example Application: main.c

```
#include <stdio.h>
#include <cpy_tbl.h>
#include <crc_tbl.h>

extern COPY_TABLE task1_ctbl;
extern COPY_TABLE task2_ctbl;
extern COPY_TABLE task3_ctbl;

extern CRC_TABLE task1_crctbl;
extern CRC_TABLE union_crctbl;

/***************************************************************************/
/* copy_in -      provided by the RTS library to copy code from its load   */
/*                address to its run address.                              */
/* my_check_CRC - verify that the CRC values stored in the given table     */
/*                match the computed value at run time, using load address. */
/* taskX -        perform a simple task.  These routines share the same run */
/*                address.                                                 */
/***************************************************************************/
extern void copy_in(COPY_TABLE *tp);
extern unsigned int my_check_CRC(CRC_TABLE *tp);
extern void task1(void);
extern void task2(void);
extern void task3(void);

int x = 0;

main()
{
   unsigned int ret_val = 0;
   unsigned int CRC_ok  = 1;

   printf("Start task copy test with CRC checking.\n");

   printf("Check CRC of task1 section.\n");
   ret_val = my_check_CRC(&task1_crctbl);
   if (ret_val == 1)
       printf("\nPASSED: CRCs for task1_crc_tbl match.\n");
   else
   {
       CRC_ok = 0;
       printf("\nFAILED: CRCs for task1_crc_tbl do NOT match.\n");
   }

   /***************************************************************************/
   /* Copy task1 into the run area and execute it.                            */
   /***************************************************************************/
   copy_in(&task1_ctbl);
   task1();

   printf("Check CRC of UNION.\n");
   if ((ret_val = my_check_CRC(&union_crctbl)) == 1)
       printf("\nPASSED: CRCs for union_crc_tbl match.\n");
   else
   {
       CRC_ok = 0;
       printf("\nFAILED: CRCs for union_crc_tbl do NOT match.\n");
```

Example C-2. Main Routine for Example Application: main.c (continued)

```
   }
   copy_in(&task2_ctbl);
   task2();
   copy_in(&task3_ctbl);
   task3();

   printf("Copy table and CRC tasks %s!!\n",
          ((CRC_ok == 1 && x == 6)) ? "PASSED" : "FAILED");
}
```

Example C-3. Checking CRC Values: check_crc.c

```
#include <stdio.h>
#include "crc_tbl.h"

/*****************************************************************************/
/* gen_crc() - computes the CRC value of data using the CRC algorithm ID    */
/*             specified.  Found in ref_crc.c                                */
/*****************************************************************************/
unsigned long gen_crc(int id, const unsigned char *data, size_t len);

/*****************************************************************************/
/* my_check_CRC() - verify the CRC values for all records stored in the     */
/*                  given CRC table.  Print diagnostic information also.    */
/*****************************************************************************/
unsigned int my_check_CRC(CRC_TABLE *tp)
{
   int i;
   unsigned int ret_val = 1;
   uint32_t my_crc;

   printf("\n\tTABLE INFO: rec size=%d, num_rec=%d.",
          tp->rec_size, tp->num_recs);

   for (i = 0; i < tp->num_recs; i++)
   {
      CRC_RECORD crc_rec = tp->recs[i];

      /************************************************/
      /* COMPUTE CRC OF DATA STARTING AT crc_rec.addr   */
      /* FOR crc_rec.size UNITS.  USE                   */
      /* crc_rec.crc_alg_ID to select algorithm.        */
      /* COMPARE COMPUTED VALUE TO crc_rec.crc_value.   */
      /************************************************/
      my_crc = gen_crc(crc_rec.crc_alg_ID, (unsigned char *)crc_rec.addr,
                       crc_rec.size);
      printf("\n\tCRC record: page=%x, alg=%x, addr = %lx, size=%lx, "
             "\n\t\tcrc=%lx, my_crc=%lx.",
             crc_rec.page_id, crc_rec.crc_alg_ID,
             crc_rec.addr, crc_rec.size, crc_rec.crc_value, my_crc);

      if (my_crc == crc_rec.crc_value)
         printf("\n\tCRCs match for record %d.\n", i);
      else
      {
         ret_val = 0;
         printf("\n\tCRCs DO NOT match for record %d.\n", i);
      }
   }
   return ret_val;
}
```

Example C-4. Task1 Routine: task1.c

```
#include <stdio.h>

extern int x;

#pragma CODE_SECTION(task1, ".task1_scn")
void task1(void) { printf("hit task1, x is %d\n", x); x += 1; }
```

Example C-5. Task2 Routine: task2.c

```
#include <stdio.h>

extern int x;

#pragma CODE_SECTION(task2, ".task2_scn")
void task2(void) { printf("hit task2, x is %d\n", x); x += 2; }
```

Example C-6. Task3 Routine: task3.c

```
#include <stdio.h>

extern int x;

#pragma CODE_SECTION(task3, ".task3_scn")
void task3(void) { printf("hit task3, x is %d\n", x); x += 3; }
```

Example C-7. Example 1 Command File: ex1.cmd (for COFF)

```
/*************************************************************************/
/* Linker Generated Copy Tables - Example #1                             */
/*                                                                       */
/* 3 separate tasks are loaded into 3 separate areas of target memory.   */
/* They all must be run in a common area of memory (overlay).  Before    */
/* each task is run, it is copied into its run space using a linker      */
/* generated copy table for each task.                                   */
/*                                                                       */
/* Two linker generated CRC tables are created.  One is for .task1_scn   */
/* and the other is for the UNION.  The UNION table will contain CRC     */
/* records for .task1_scn, .task2_scn and .task3_scn.                    */
/*************************************************************************/
-c
-x

ex1.obj
task1.obj
task2.obj
task3.obj
check_crc.obj
ref_crc.obj

-o ex1.out
-m ex1.map

-stack    0x1000
-heap     0x800

MEMORY
{
PAGE 0 : RESET(R):    origin = 0x000000, length =  0x00002
```

Example C-7. Example 1 Command File: ex1.cmd (for COFF) (continued)

```
         VECTORS(R) : origin = 0x000002, length =  0x003FE
         PROG(R)    : origin = 0x3f0000, length =  0x10000
PAGE 1 : RAM1 (RW)  : origin = 0x000402 , length = 0x003FE
PAGE 1 : RAM2 (RW)  : origin = 0x001000 , length = 0x04000
PAGE 1 : RAM3 (RW)  : origin = 0x3e0000 , length = 0x08000
}

SECTIONS
{
   UNION
   {
      .task2_scn: load = RAM3, PAGE = 1, table(_task2_ctbl)
      .task3_scn: load = RAM3, PAGE = 1, table(_task3_ctbl)
      .task1_scn: load = RAM3, PAGE = 1, table(_task1_ctbl),
                                         crc_table(_task1_crctbl)
   } run = PROG, PAGE = 0, crc_table(_union_crctbl, algorithm=CRC16_ALT)

   vectors  : load = VECTORS, PAGE = 0
   .text    : load = PROG, PAGE = 0
   .data    : load = 440h, PAGE = 1
   .cinit   : > PROG, PAGE = 0
   .ebss    : > RAM3, PAGE = 1
   .econst  : > RAM3, PAGE = 1
   .reset   : > RESET, PAGE = 0
   .stack   : > RAM2, PAGE = 1
   .sysmem  : > RAM2, PAGE = 1
   .esysmem : > RAM3, PAGE = 1

   .ovly     > RAM2, PAGE = 1
}
```

Glossary

D.1 Terminology

ABI — Application binary interface.

absolute address — An address that is permanently assigned to a TMS320C28x memory location.

absolute constant expression — An expression that does not refer to any external symbols or any registers or memory reference. The value of the expression must be knowable at assembly time.

absolute lister — A debugging tool that allows you to create assembler listings that contain absolute addresses.

address constant expression — A symbol with a value that is an address plus an addend that is an absolute constant expression with an integer value.

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. For TMS320C28x, the size of a byte is 16-bits, which is also the size of a word.

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.

constant expression — An expression that does not in any way refer to a register or memory reference.

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).

DWARF — A standardized debugging data format that was originally designed along with ELF, although it is independent of the object file format.

EABI — An embedded application binary interface (ABI) that provides standards for file formats, data types, and more.

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 TMS320C28x 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 TMS320C28x, 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.

identifier— Names used as labels, registers, and symbols.

immediate operand — An operand whose value must be a constant expression.

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).

literal constant — A value that represents itself. It may also be called a *literal* or an *immediate value*.

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.

memory reference operand — An operand that refers to a location in memory using a target-specific syntax.

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.

register operand — A special pre-defined symbol that represents a CPU register.

relocatable constant expression— An expression that refers to at least one external symbol, register, or memory location. The value of the expression is not known until link time.

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 TMS320C28x 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 name that represents an address or a value.

symbolic constant — A symbol with a value that is an absolute constant expression.

symbolic debugging — The ability of a software tool to retain symbolic information that can be used by a debugging tool such as an emulator or simulator.

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

Revision History

E.1 Recent Revisions

This table lists significant changes made to this document. The left column identifies the first version of this document in which a particular change appeared.

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRU513S		-- throughout --	The default file extensions for object files created by the compiler have been changed in order to prevent conflicts when C and C++ files have the same names. Object files generated from C source files have the .c.obj extension. Object files generated from C++ source files have the .cpp.obj extension. Object files generated from assembly source files still have the .obj extension.
SPRU513S	Hex Conversion Utility	Section 12.12	Added support for the secure flash boot capability provided by TMS320F2838x devices.
Previous Revisions:			
SPRU513R.1	Linker	Section 8.5.9	Added documentation for the NOINIT special section type. (EABI only)
SPRU513R		-- throughout --	Added further documentation of EABI support. This includes identifying features that are supported only for COFF or EABI. Identified examples as COFF-specific where necessary.
SPRU513R	Object Modules	Section 2.1	Added information about the ELF object file format.
SPRU513R	Object Modules	Section 2.3.1, Section 2.4.1	Added information about section names used by EABI.
SPRU513R	Object Modules	Section 2.6	Revised information about types of symbols for clarity.
SPRU513R	Program Loading	Section 3.3.2	Added information about the RAM model and ROM model for EABI.
SPRU513R	Assembler Description	Section 4.4	Added information about controlling the ABI setting.
SPRU513R	Assembler Description	Section 4.8.6	Corrected list of symbolic constants.
SPRU513R	Assembler Description	Section 4.8.7	Added the documentation for the FPU RB register.
SPRU513R	Assembler Directives	Section 5.12	Added topics for directives related to EABI: .bss, .common, .elfsym, .group, .gmember, endgroup, .retain, .retainrefs, .weak, .xfloat, and .xldouble.
SPRU513R	Assembler Directives	.bits topic	Modified the description of the .bits directive.
SPRU513R	Assembler Directives	.usect topic	Modified the description of the .usect directive.
SPRU513R	Assembler Directives	.symdepend topic, .weak topic	Split .symdepend and .weak directive topics.
SPRU513R	Linker	Section 8.4	Added the --emit_references:file linker option.
SPRU513R	Linker	Section 8.4	Added linker options related to EABI: --cinit_compression, --copy_compression, --retain, --unused_section_elimination, --warn_sections, and --zero_init.
SPRU513R	Linker	Section 8.5.11.4	Added symbols automatically defined by the linker for EABI.
SPRU513R	Linker	Section 8.6.2	Added information about weak symbols used by EABI.
SPRU513R	Linker	Section 8.8.5	Added information about compression available with EABI.

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRU513Q			Added support for EABI. The COFF ABI is the default.
SPRU513Q	Assembler, Linker	Section 4.8.6, Section 4.11, and Section 8.4.11	Added support for 64-bit floating point operations (--float_support=fpu64). Added support for fast integer division (--idiv_support=idiv0). Added support for additional TMU instructions (--tmu_support=tmu1). Added support for Cyclic Redundancy Check (CRC) with the VCU (--vcu_support=vcrc).
SPRU513Q	Linker	Section 8.4, Section 8.4.12, and Section 8.5.10	Added the --ecc=on linker option, which enables ECC generation. Note that ECC generation is now off by default.
SPRU513Q	Linker	Section 8.5.7.3	Added linker syntax to combine initialized section with uninitialized sections.
SPRU513P	Hex Conversion Utility	Section 12.2.1 and Section 12.10	Added the --array option, which causes the array output format to be generated.
SPRU513M	Object Modules, Assembler Directives	Section 2.4.1 and .usect topic	Added information about DP load optimization.
SPRU513M	Assembler Description	Section 4.3, Section 4.11, and Section 4.11.3	Documented support for CLA version 2 and CLA v2 background tasks.
SPRU513M	Assembler Directives	.usect topic	Explain the effect of the alignment flag for the .usect directive.
SPRU513M	Linker Description	Section 8.9	Provided a link to an E2E blog post that provides examples that perform cyclic redundancy checking using linker-generated CRC tables.
SPRU513L	Linker Description	Section 8.5.10	Documented revised behavior of ECC directives.
SPRU513K	Linker Description	Section 8.4	Several linker options have been deprecated, removed, or renamed. The linker continues to accept some of the deprecated options, but they are not recommended for use. See the Compiler Option Cleanup wiki page for a list of deprecated and removed options, options that have been removed from CCS, and options that have been renamed.
SPRU513J	Linker Description	Section 8.5.3	Information about accessing files and libraries from a linker command file has been added.
SPRU513J	Linker Description	Section 8.9.2	The list of available CRC algorithms has been expanded.
SPRU513J	Object File Utilities	Section 11.1	A –cg option has been added to the Object File Display utility to display function stack usage and callee information in XML format.
SPRU513I	Program Loading, Linker	Section 3.3.3.1 and Section 8.8.4.2	Added the BINIT (boot-time initialization) copy table.
SPRU513I	Linker	Section 8.4.20	Added modules as a filter for the --mapfile_contents linker option.
SPRU513I	Linker	Section 8.5.5.2.1	Added an example for placing functions in RAM.
SPRU513I	Linker	Section 8.8.4.3	Documented the table() operator.
SPRU513H	--	--	The near and far keywords are deprecated, and the small memory model is no longer supported; the only memory model uses 32-bit pointers. The C27x object mode is also no longer supported. The .bss, .const, and .sysmem sections are no longer used; the .ebss, .econst, and .esysmem sections are used instead. As a result, the --farheap linker option, far call trampolines, and several other related features are no longer documented.
SPRU513H	Object Modules	Section 2.4.4	Added information about the current section and how directives interact with it.
SPRU513H	Object Modules	Section 2.6 and Section 2.6.4	Added information about various types of symbols and about symbol tables.
SPRU513G	Assembler Description	Section 4.3 and Section 4.11	Added support for Type 2 VCU via --vcu_support=vcu2.
SPRU513G	Assembler Description	Section 4.3 and Section 4.11	Added support for Type 1 CLA via --cla_support=cla1.
SPRU513H	Assembler Description	Section 4.11.3	The naming of function frames in scratchpad memory for the CLA compiler has changed.

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRU513H	Linker	Section 8.4.2, Section 8.5.11.7, and Section 8.6.1	Added information about referencing linker symbols.
SPRU513H	Linker	Section 8.4.11	Added a list of the linker's predefined macros.
SPRU513G	Linker	Section 8.5.5.1	Removed invalid syntax for load and fill properties.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

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