TMS320C28x Assembly Language Tools v5.0.0

User's Guide

Literature Number: SPRU513C
October 2007

Texas Instruments

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 these assembly language tools:

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

Notational Conventions

This document uses the following conventions:

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

  Here is a sample of C code:

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

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

  cl2000 -v28 [*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 -v28 --run_linker {**--rom_model | --ram_model**} *filenames* [**--output_file=***name.out*]
  --library= *libraryname*

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

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

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. This syntax is shown as [, ..., *parameter*].

Related Documentation From Texas Instruments

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

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

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 tthe central processing unit (CPU) and the assembly language instructions of the TMS320C28x™ fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

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

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

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
- Link step
- Library information archiver
- Absolute lister
- Cross-reference lister
- Object file display utility
- Name utility
- Strip utility
- Hex conversion utility

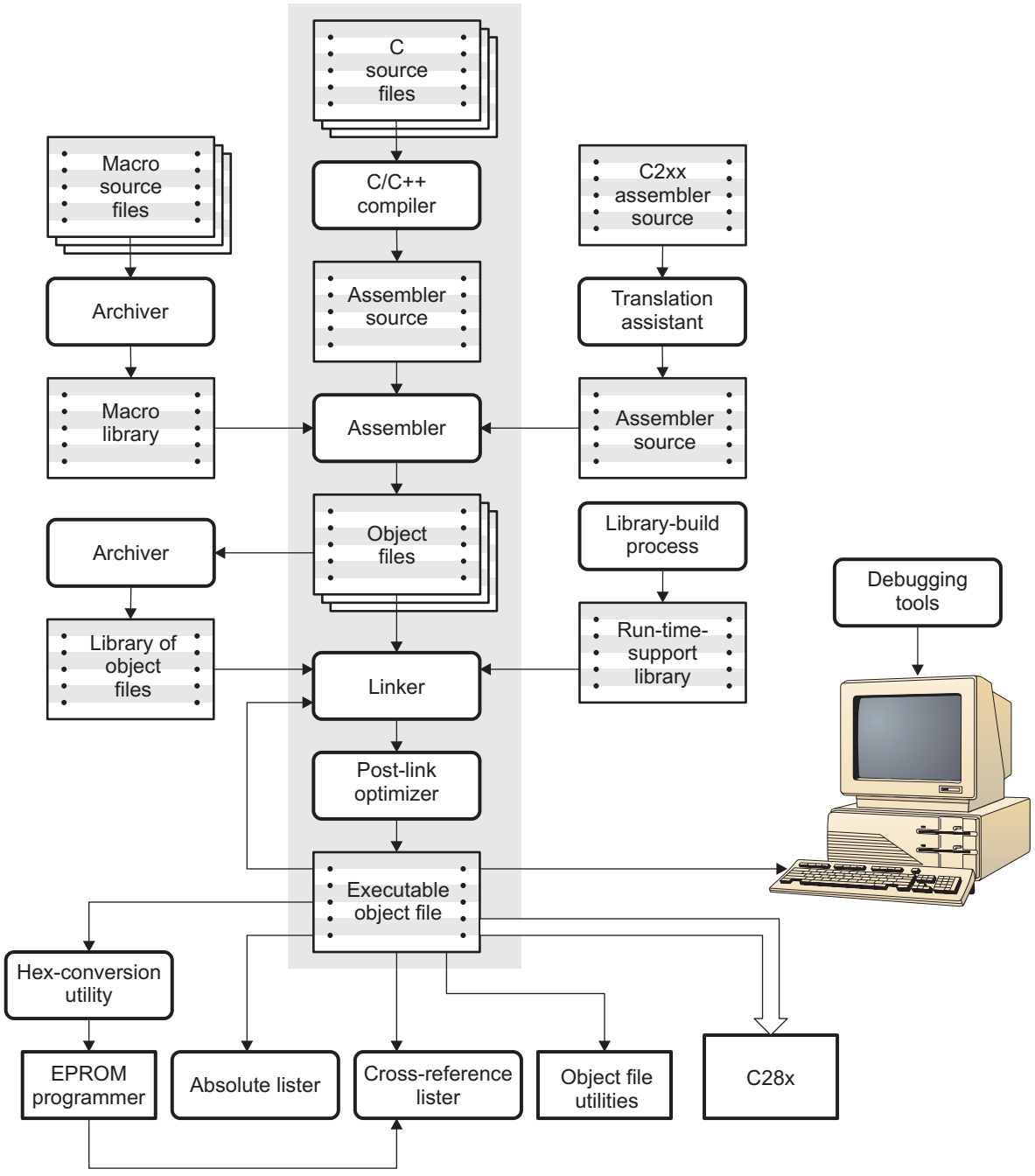
This chapter shows how these tools fit into the general software tools development flow and gives a brief description of each tool. For convenience, it also summarizes the C/C++ compiler and debugging tools. For detailed information on the compiler and debugger, and for complete descriptions of the TMS320C28x, refer to 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 assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:
  - The shell program enables you to compile, assemble, and link source modules in one step.
  - The optimizer modifies code to improve the efficiency of C/C++ programs.
  - The interlist utility interlists C/C++ source statements with assembly language output to correlate code produced by the compiler with your source code.

  See the *TMS320C28x C/C++ Compiler User's Guide* for more information.
- 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 various aspects of the assembly process, such as the source listing format, data alignment, and section content. See Chapter 3 through Chapter 5. See the *TMS320C28x DSP CPU and Instruction Set Reference Guide* for detailed information on the assembly language instruction set.
- The **link step** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable object modules (created by the assembler) as input. It also accepts archiver library members and output modules created by a previous linker run. Link directives allow you to combine object file sections, bind sections or symbols to addresses or within memory ranges, and define or redefine global symbols. See Chapter 7.
- The **archiver** allows you to collect a group of files into a single archive file, called a library. For example, you can collect several macros into a macro library. The assembler searches the library and uses the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members. See Chapter 6.
- You can use the **library-build process** to build your own customized run-time-support library. See the *TMS320C28x C/C++ Compiler User's Guide* for more information.
- The **hex conversion utility** converts an object file into TI-Tagged, ASCII-Hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer. See Chapter 11.
- The **absolute lister** uses linked object files to create .abs files. These files can be assembled to produce a listing of the absolute addresses of object code. See Chapter 8.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definition, and their references in the linked source files. See Chapter 9.
- The main product of this development process is a module that can be executed in 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:

- The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both human readable and XML formats. See Section 10.1.
- The **name utility** prints a list of names defined and referenced in a object or an executable file. See Section 10.2.
- The **strip utility** removes symbol table and debugging information from object and executable files. See Section 10.3.

Introduction to Object Modules

The assembler and link step create object modules that 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 link step provide directives that allow you to create and manipulate sections.

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

Topic **Page**

2.1 Sections

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

.text section	usually contains executable code
.data section	usually contains initialized data
.bss section	usually reserves space for uninitialized variables

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

There are two basic types of sections:

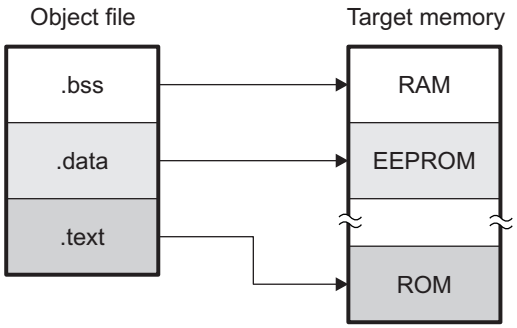
Initialized sections	contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.
Uninitialized sections	reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created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

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

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

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

Figure 2-1. Partitioning Memory Into Logical Blocks



2.2 How the Assembler Handles Sections

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

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

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

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

Default Sections Directive

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

2.2.1 Uninitialized Sections

Uninitialized sections reserve space in TMS320C28x memory; they are usually allocated into RAM. These sections have no actual contents in the object file; they simply reserve memory. A program can use this space at run time for creating and storing variables.

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

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

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

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

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

size in words is an absolute expression. The .bss directive reserves *size in words* words in the .bss section. The .usect directive reserves *size in words* words in *section name*. For either directive you must specify a size; there is no default value.

blocking flag is an optional parameter. If you specify a value greater than 0 for this parameter, the assembler allocates *size in words* contiguously. This means the allocated space does not cross a page boundary unless its size is greater than a page, in which case the objects starts a page boundary.

alignment flag is an optional parameter. It causes the assembler to allocate *size in words* on long word boundaries.

type is an optional parameter. It causes the assembler to produce the appropriate debug information for the symbol. See Section 3.15 for more information.

section name tells the assembler which named section to reserve space in. The section name must be enclosed in quotation marks. See Section 2.2.3.

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

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

2.2.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in TMS320C28x memory when the program is loaded. Each initialized section is independently relocatable and may reference symbols that are defined in other sections. The link step automatically resolves these section-relative references.

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

.text

.data

.sect " *section name* **"**

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

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

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

2.2.3 Named Sections

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

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

Two directives let you create named sections:

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

The syntaxes for these directives are:

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

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

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

2.2.4 Subsections

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

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

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

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

Using the link step's SECTIONS directive, you can allocate .text:_func separately, or with all the .text sections. See Section 7.8.1 for an example using subsections.

You can create two types of subsections:

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

Subsections are allocated in the same manner as sections. See Section 7.8 for information on the SECTIONS directive.

2.2.5 Section Program Counters

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

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

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

2.2.6 Using Sections Directives

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

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

Field 1 contains the source code line counter.

Field 2 contains the section program counter.

Field 3 contains the object code.

Field 4 contains the original source statement.

See Section 3.13 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 .bss 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 file in Figure 2-2 creates four sections:

.text contains ten 32-bit words of object code.

.data contains five words of initialized data.

.bss reserves ten words in memory.

newvars is a named section created with the .usect directive; it contains eight words in memory.

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

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

Line 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		

2.3 How the Link Step Handles Sections

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

Two link step 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 link step how to combine input sections into output sections and where to place these output sections in memory.

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

It is not always necessary to use link step directives. If you do not use them, the link step uses the target processor's default allocation algorithm described in Section 7.13. When you *do* use link step directives, you must specify them in a link step command file.

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

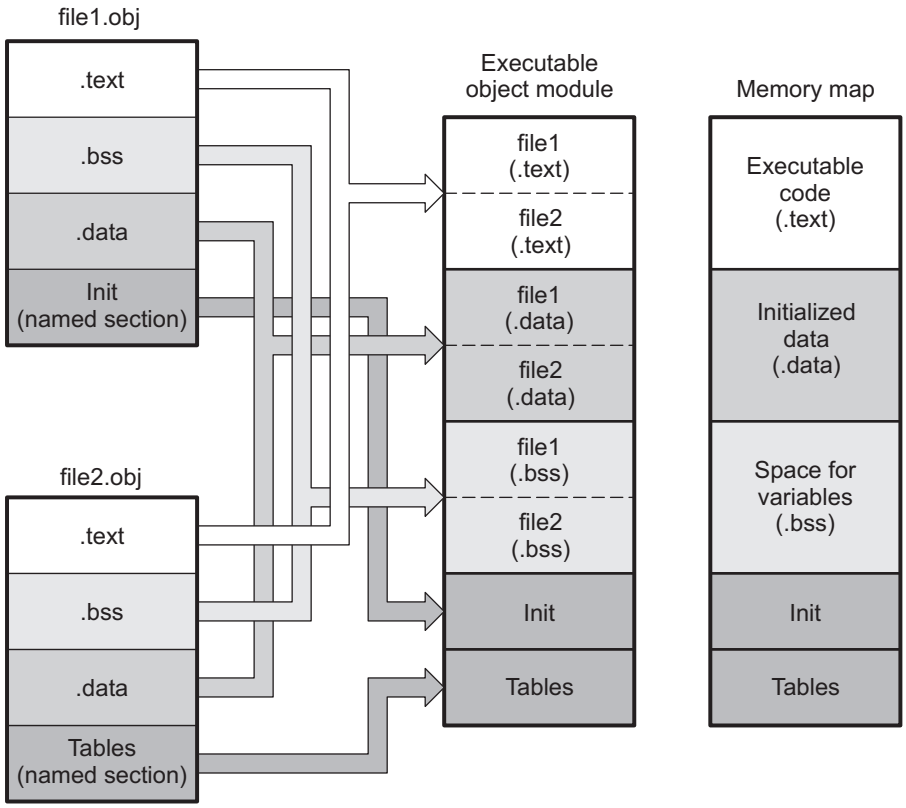
- Section 7.5, *Link Step Command Files*
- Section 7.7, *The MEMORY Directive*

- Section 7.8, *The SECTIONS Directive*
- Section 7.13, *Default Allocation Algorithm*

2.3.1 Default Memory Allocation

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

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



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

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

The C/C++ compiler uses the .const section to store string constants, and variables or arrays that are declared as *const*. The C/C++ compiler produces tables of data for autoinitializing global variables; these variables are stored in a named section called .cinit (see Example 7-7). For more information on the .const and .cinit sections, see the *TMS320C28x C/C++ Compiler User's Guide*.

2.3.2 Placing Sections in the Memory Map

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

For further explanation of section placement within the memory map, see the discussions in Section 7.7 and Section 7.8.

2.4 Relocation

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

- Allocating them into the memory map so that they begin at the appropriate address as defined with the link step'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 link step uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The link step then uses these entries to patch the references after the symbols are relocated. Example 2-1 contains a code segment for a TMS320C28x device that 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
```

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 link step uses the two relocation entries to patch the two references in the object code:

```
0080' LC Y              becomes     0080'
0004                                7204
28A1! MOV AR1,#X        becomes     28A1!
0000                                7100
```

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 link step 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 link step relocates sym2 to 300h and sym1 to 200h. Then the link step 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
```

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 link step.

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 link step usually removes relocation entries after it uses them. This prevents the output file from being relocated again (if it is relinked or when it is loaded). A file that contains no relocation entries is an *absolute* file (all its addresses are absolute addresses). If you want the link step to retain relocation entries, invoke the link step with the --relocatable option (see Section 7.4.1.2).

2.5 Run-Time Relocation

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

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

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

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

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

For a complete description of run-time relocation, see Section 7.9.

2.6 Loading a Program

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

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

- The TMS320C28x debugging tools, including the simulator, have built-in loaders. Each of these tools contains a LOAD command that invokes the loader. The loader reads the executable file and copies the program into target memory.
- You can use the hex conversion utility (hex2000, which is shipped as part of the assembly language package) to convert the executable object module into one of several object file formats. You can then use the converted file with an EPROM programmer to burn the program into an EPROM.

2.7 Symbols in an Object File

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

2.7.1 External Symbols

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

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

The following code segment illustrates these definitions.

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

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

a:      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 link step to look through the symbol tables of other files for y's and z's definitions.

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

2.7.2 The Symbol Table

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

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

Assembler Description

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

Assembler directives	described in Chapter 4
Macro directives	described in Chapter 5
Assembly language instructions	described in the *TMS320C28x DSP CPU and Instruction Set Reference Guide*

Topic **Page**

3.1 Assembler Overview

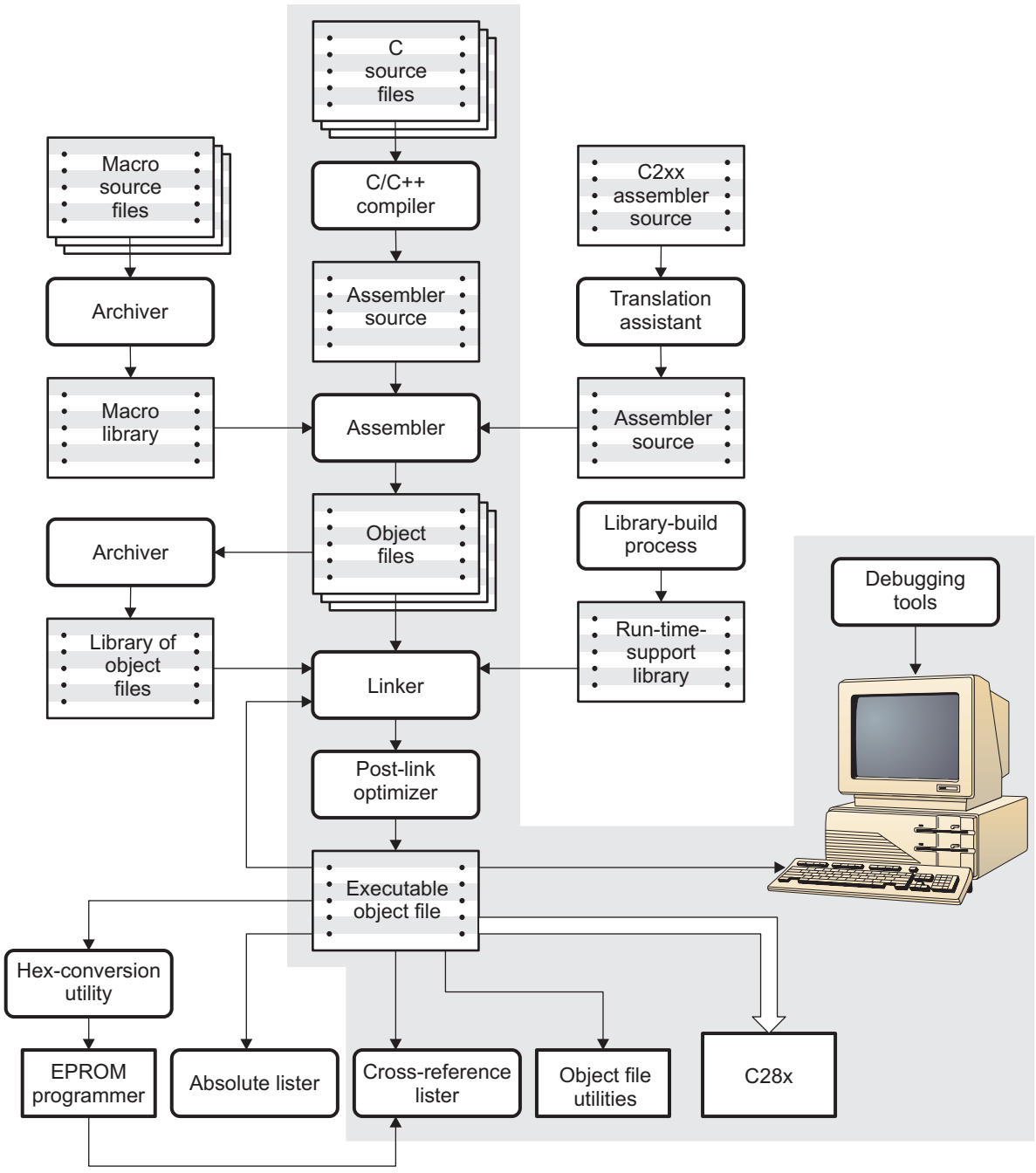
The 2-pass assembler does the following:

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

3.2 The Assembler's Role in the Software Development Flow

Figure 3-1 illustrates the assembler's role in the software development flow. The shaded portion highlights the most common assembler development path. The assembler accepts assembly language source files as input, both those you create and those created by the TMS320C28x C/C++ compiler.

Figure 3-1. The Assembler in the TMS320C28x Software Development Flow



3.3 Invoking the Assembler

To invoke the assembler, enter the following:

cl2000 *version 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 calls the assembler.

version refers to the target processor for which the source file is assembled. The valid versions are -v28 for the TMS320C28x processor and -v27 for the TMS320C27x processor. The version is required; the assembler issues an error if version is not specified.
If both -v27 and -v28 are specified, the assembler ignores the second version and issues a warning. For more information, see Section 3.12.

input file names the assembly language source file.

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

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

Table 3-1. TMS320C28x Assembler Options

Option	Alias	Description
--absolute_listing	**-aa**	creates an absolute listing. When you use -aa, the assembler does not produce an object file. The -aa option is used in conjunction with the absolute lister.
--asm_code_fill		specifies fill value for code sections. Default is zero. See Section 3.11.
--asm_data_fill		specifies fill value for data sections. Default is NOP instructions. See Section 3.11.
--asm_define=*name*[*=def*]	**-ad**	sets the *name* symbol. This is equivalent to inserting *name* **.set** [*value*] at the beginning of the assembly file. If *value* is omitted, the symbol is set to 1. See Section 3.8.4.
--asm_dependency	**-apd**	performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a .ppa extension.
--asm_includes	**-api**	performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of files included with the .include directive. The list is written to a file with the same name as the source file but with a .ppa extension.
--asm_listing	**-al**	produces a listing file with the same name as the input file with a *.lst* extension.
--asm_remarks	**-mw**	enables additional assembly-time checking. A warning is generated if a .bss allocation size is greater than 64 words, or a 16-bit operand value resides outside of the -32768 to 65535 range.
--asm_undefine= *name*	**-au**	undefines the predefined constant *name*, which overrides any -ad options for the specified constant.
--c2xlp_src_compatible	**-m20**	accepts C2xLP assembly instructions and encodes them as equivalent C28x instructions. The --c2xlp_src_compatible option implies the -v28 option. See Section 3.12.
--cdebug_asm_data	**-mg**	produces C-type symbolic debugging for assembly variables defined in assembly source code using data directives. This support is for basic C types, structures, and arrays.
--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"

Table 3-1. TMS320C28x Assembler Options (continued)

Option	Alias	Description
--copy_file= *filename*	**-ahc**	copies the specified file for the assembly module. The file is inserted before source file statements. The copied file appears in the assembly listing files.
--cross_reference	**-ax**	produces a cross-reference table and appends it to the end of the listing file; it also adds cross-reference information to the object file for use by the cross-reference utility. If you do not request a listing file but use the -ax option, the assembler creates a listing file automatically, naming it with the same name as the input file with a *.lst* extension.
--disable_pcd		disables pipeline conflict detection in the assembler. For floating point unit (FPU) target only.
--float_support={ fpu32|fpu64}		assembles code for C28x with 32-bit or 64-bit hardware FPU support. This option requires the –v28 option and assumes the large memory model (--large_memory_model option) is specified.
--include_file= *filename*	**-ahi**	includes the specified file for the assembly module. The file is included before source file statements. The included file does not appear in the assembly listing files.
--include_path= *pathname*	**-I**	specifies a directory where the assembler can find files named by the .copy, .include, or .mlib directives. There is no limit to the number of directories you can specify in this manner; each pathname must be preceded by the --include_path option. See Section 3.4.1.
--large_memory_model	**-mf**	allows conditional compilation of 16-bit code with large memory model code. Defines the LARGE_MODEL symbol and sets it to true.
--out_as_uout	**-mu**	encodes C2xlp OUT instructions as C28x UOUT instructions. The C28x processor has protected (OUT) and unprotected (UOUT) instructions. By default, the assembler encodes C2xlp OUT instructions as C28x protected OUT instructions. This option is ignored if --c2xlp_src_compatible is not specified.
--output_all_syms	**-as**	puts all defined symbols in the object file's symbol table. The assembler usually puts only global symbols into the symbol table. When you use -as, symbols defined as labels or as assembly-time constants are also placed in the table.
--quiet	**-q**	suppresses the banner and progress information (assembler runs in quiet mode).
--symdebug:dwarf	**-g**	enables assembler source debugging in the C source debugger. Line information is output to the object module for every line of source in the assembly language source file. You cannot use the --symdebug:dwarf option on assembly code that contains .line directives. See Section 3.14.
--syms_ignore_case	**-ac**	makes case insignificant in the assembly language files. For example, --syms_ignore_case makes the symbols ABC and abc equivalent. *If you do not use this option, case is significant* (default). Case significance is enforced primarily with symbol names, not with mnemonics and register names.

3.4 Naming Alternate Directories for Assembler Input

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

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

The *filename* names a copy/include file that the assembler reads statements from or a macro library that contains macro definitions. If *filename* begins with a number the double quotes are required. 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

Because of this search hierarchy, you can augment the assembler's directory search algorithm by using the --include_path option (described in Section 3.4.1) or the C2000_A_DIR environment variable (described in Section 3.4.2).

3.4.1 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 -v28 --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 -v28 --include_path=/tools/files source.asm`
Windows	`cl2000 -v28 --include_path=c:\tools\files source.asm`

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

3.4.2 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 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 C2000_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 C2000_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 -v28 --include_path=/tools/files source.asm`
Windows	`set C2000_A_DIR=c:\dsys` `cl2000 -v28 --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)	`set C2000_A_DIR=`
Windows	`unset C2000_A_DIR`

3.5 Source Statement Format

TMS320C28x assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. A source statement can contain five ordered fields (label, mnemonic, unit specifier, operand list, and comment). The general syntax for source statements is as follows:

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

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

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. Tab and space characters are blanks. You must separate the operand list from the preceding field with a blank.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column *must* begin with a semicolon.
- A mnemonic cannot begin in column 1 or it will be interpreted as a label.

The following sections describe each of the fields.

3.5.1 Label Field

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

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

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

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

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

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

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

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

3.5.2 Mnemonic Field

The mnemonic field follows the label field. The mnemonic field cannot start in column 1; if it does, it is interpreted as a label. 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
```

The mnemonic field can begin with 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 call

3.5.3 Operand Field

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

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

You must separate operands with commas.

3.5.4 Comment Field

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

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

3.6 Constants

The assembler supports seven types of constants:

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

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

3.6.1 Binary Integers

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

00000000B	Constant equal to 0_{10} or 0_{16}
0100000b	Constant equal to 32_{10} or 20_{16}
01b	Constant equal to 1_{10} or 1_{16}
11111000B	Constant equal to 248_{10} or $0F8_{16}$

3.6.2 Octal Integers

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

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

Octal Numbers Are Not Accepted With C2xlp Syntax Mode

Note: When the -v28 --c2xlp_src_compatible options are specified, cl2000 accepts C2xlp source code. The C2xlp assembler interpreted numbers with leading zeros as decimal integers, that is 010 was interpreted as 10. Because of this, when cl2000 is invoked with -v28 --c2xlp_src_compatible --asm_remarks, the assembler issues a warning when it encounters an octal number.

3.6.3 Decimal Integers

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

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

3.6.4 Hexadecimal Integers

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

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

3.6.5 Character Constants

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

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

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

3.6.6 Assembly-Time Constants

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

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

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

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

3.6.7 Floating-Point Constants

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

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

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

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

3.7 Character Strings

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

These are examples of valid character strings:

"sample program"	defines the 14-character string *sample program.*
"PLAN ""C"""	defines the 8-character string *PLAN "C".*

Character strings are used for the following:

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

3.8 Symbols

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

3.8.1 Labels

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

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

```
        .global label1

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

3.8.2 Local Labels

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

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

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

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

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

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

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

```
$1:
      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 can be in effect at one time. After you undefine a local label, you can define it and use it again. Local labels do not appear in the object code symbol table.

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

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

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

3.8.3 Symbolic Constants

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

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

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

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

The assembler also has several predefined symbolic constants; these are discussed in Section 3.8.5.

3.8.4 Defining Symbolic Constants (--asm_define Option)

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

cl2000 -v28 --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 Studio, enter the definition in a file and include that file with the -@ option.

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

```
cl2000 -v28 --asm_define=SYM1=1 --asm_define=SYM2=2 --asm_define=SYM3=3 --asm_define=SYM4=4
value.asm
```

Since you have assigned values to SYM1, SYM2, SYM3, and SYM4, you can use them in source code. Example 3-3 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 the following directives:

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

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

Example 3-3. 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
```

3.8.5 Predefined Symbolic Constants

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

- **$**, the dollar-sign character, represents the current value of the section program counter (SPC). $ is a relocatable symbol.
- **Processor symbols**, including the following:

Symbol name	Description
	Always set to 1
.TMS320C2700	Set to 1 for C27x (-v27 option), otherwise 0
.TMS320C2800	Set to 1 for C28x (-v28 option), otherwise 0
.TMS320C2800_FPU32	Set to 1 for C28x (-v28 option) with 32-bit FPU support, otherwise 0
__LARGE_MODEL	Set to 1 for large model mode (-mf option); otherwise 0

- **CPU control registers**, including the following:

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

Register	Description
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/muliplicant high, multiplicant low

Control registers can be entered as all upper-case or all lower-case characters; for example, IER can also be entered as ier.

- **FPU control registers**, including the following:

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

3.8.6 Substitution Symbols

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

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

```
    .asg    "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 5 for more information about macros.

3.9 Expressions

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

Parentheses
Expressions enclosed in parentheses are always evaluated first.
8 / (4 / 2) = 4, but 8 / 4 / 2 = 1
You *cannot* substitute braces ({ }) or brackets ([ ]) for parentheses.

Precedence groups
Operators, listed in Table 3-2, are divided into nine precedence groups. When parentheses do not determine the order of expression evaluation, the highest precedence operation is evaluated first.
8 + 4 / 2 = 10 (4 / 2 is evaluated first)

Left-to-right evaluation
When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated from left to right, except for Group 1, which is evaluated from right to left.
8 / 4*2 = 4, but 8 / (4*2) = 1

3.9.1 Operators

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

Differences in Precedence From Other TMS320 Assemblers

Notes:

- Some TMS320 assemblers use a different order of precedence than the TMS320C28x assembler uses. For this reason, different results may be produced from the same source code. The TMS320C28x uses the same order of precedence that the C language uses.
- When cl2000 is invoked with --silicon_version=v28 --c2xlp_src_compatible, the assembler accepts C2xlp source code. A programmer writing code for the C2xlp assembler would assume different precedence than that used by the C28x assembler. Therefore when invoked with the --silicon_version=v28 --c2xlp_src_compatible --asm_remarks options, the C28x assembler issues a warning when it encounters an expression such as a + b << c.

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

Group[1]	Operator	Description[2]
1	+	Unary plus
	-	Unary minus
	~	1s complement
	!	Logical NOT
2	*	Multiplication
	/	Division
	%	Modulo
3	+	Addition
	-	Subtraction
4	<<	Shift left
	>>	Shift right
5	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to

(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.

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

Group[1]	Operator	Description[2]
6	=[=] !=	Equal to Not equal to
7	&	Bitwise AND
8	^	Bitwise exclusive OR (XOR)
9	|	Bitwise OR

3.9.2 Expression Overflow and Underflow

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

3.9.3 Well-Defined Expressions

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

This is an example of a well-defined expression:

```
1000h+X
```

where X was previously defined as an absolute symbol.

3.9.4 Conditional Expressions

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

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

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

3.9.5 Legal Expressions

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

When an expression contains more than one relocatable symbol or cannot be evaluated at assembly time, the assembler encodes a relocation expression in the object file that is later evaluated by the linker. If the final value of the expression is larger in bits than the space reserved for it, you receive an error message from the linker. See Section 2.4 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 3.9.3. For example:

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

3.9.6 Expression Examples

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

```
          .global extern_1 ; Defined in an external module
intern_1: .word '"10'      ; Relocatable, defined in
                           ;   current module
LAB1:     .set 2           ; LAB1 = 2
intern_2                   ; Relocatable, defined in
                           ;   current module
intern_3                   ; Relocatable, defined in
                           ;   current module
```

- **Example 1**

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

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

- **Example 2**

  All of the following statements are valid.

```
MOV @(extern_1 - 10), AL        ; Legal
MOV @(10-extern_1),   AL        ; Legal
MOV @(-(intern_1)), AL          ; Legal
MOV @(extern_1/10), AL          ; / not an additive operator
MOV @(intern_1 + extern_1),ACC  ; Multiple relocatables
```

- **Example 3**

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

```
MOV (intern_1 - intern_2 + extern_1), ACC    ; Legal
MOV (intern_1 + intern_2 + extern_1), ACC    ; Illegal
```

- **Example 4**

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

```
MOV (intern_1 + extern_1 - intern_2), ACC    ; Illegal
```

3.10 Built-In 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 3-3 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 3-3. Built-In Mathematical Functions

Function	Description
$acos(*x*)	Returns $\cos^{-1}(x)$ in range [0, π], *x* [-1, 1]
$asin(*x*)	Returns $\sin^{-1}(x)$ in range [-π/2, π/2], *x* [-1, 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
$tan(*x*)	Returns the tangent of *x*
$tanh(*x*)	Returns the hyperbolic tangent of *x*
$trunc(*x*)	Returns *x* truncated toward 0

The built-in substition symbol functions are discussed in Section 5.3.2.

3.11 Specifying Assembler Fill Values (--asm_code_fill and --asm_data_fill)

The C28x assembler allows you to specify fill values to fill the holes created by the assembler.

The .align directive aligns the section program counter (SPC) on the next boundary, depending on the size in words parameter. The assembler might create holes to align the SPC. The assembler uses the default values of zero for data sections and NOP instructions for code sections.

A code section is defined as either a .text section or any section that has an instruction to encode. The following are considered to be code sections:

Example 1

```
.text
.field 0x100, 16
```

Example 2

```
.data
.field 0x100, 16
 MOV al, #1
```

Example 3

```
.sect "MyProg"
MOV   al, #0
```

Any section other than a .text section is considered a data section if it does not have any instruction to encode. For example:

```
.sect "MyData"
.field 0x100, 16
```

The assembler supports the --asm_code_fill and --asm_data_fill options to enable you to specify the fill values for the code sections and data sections, respectively. You can specify a 16-bit value with these options in decimal (4660), octal (011064) or hexadecimal (0x1234) format. For example consider the following assembly code:

```
.sect   "MyData"
.align  1
.field          0x01,16
.align  2
.field          0x00010002,32

.sect   "MyProg"
.align  1
MOV     ah, #0
.align  2
MOV     acc,#1234 << 5
```

3.12 TMS320C28x Assembler Modes

The TMS320C28x processor is object code compatible with the TMS320C27x processor and source code compatible with the TMS320C2xx (C2xlp) processor. The C28x assembler operates in four different modes to support backward compatibility with C27x and C2xlp processors. These four modes are controlled by the options as follows:

-v27	C27x object mode
-v28	C28x object mode
-v28 --c2xlp_src_compatible	C28x object mode--Accept C2xlp Syntax Mode

The --c2xlp_src_compatible option implies the -v28 (or --silicon_version=28) option. Therefore you do not need to specify -v28 explicitly.

When multiple versions are specified, the assembler uses the first version specified and ignores the rest. For example the command cl2000 -v28 -v27 invokes the assembler in the C28x object mode and the assembler ignores the -v27 switch. Also the assembler issues the following warning:

```
>> Version already specified. -v27 is ignored
```

Since --c2xlp_src_compatible implies the version -v28 the command cl2000 --c2xlp_src_compatible -v27 is equivalent to cl2000 -v28 --c2xlp_src_compatible -v27. Therefore the assembler generates the above warning and ignores the -v27 switch.

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

To support some special floating point instructions when a 32-bit floating point unit (FPU) is available, the assembler operates in FPU32 mode. Section 3.12.5 describes the FPU32 mode. This mode is controlled by options as follows.

-v28 --float_support=fpu32 C28x object mode--Accept FPU32 instructions

3.12.1 C27x Object Mode

This mode is used to port C27x code to the C28x and run the C28x processor in C27x object mode. The C28x assembler in this mode is essentially the C27x assembler that supports the following non-C27x instructions. These instructions are used for changing the processor object mode and addressing modes. Refer to the *TMS320C28x DSP CPU and Instruction Set Reference Guide* for more details on these instructions.

Table 3-4. Non-TMS320C27x Instructions Supported in the C27x Object Mode

Instructions	Description
SETC OBJMODE	Set the OBJMODE bit in the status register. The processor runs in the C28x object mode.
CLRC OBJMODE	Clear the OBJMODE bit in the status register. The processor runs in the C27x object mode.
C28OBJ	Same as SETC OBJMODE
C27OBJ	Same as CLRC OBJMODE
SETC AMODE	Set the AMODE bit in the status register. The processor supports C2xlp addressing.
CLRC AMODE	Clear the AMODE bit in the status register. The C28x processor supports C28x addressing.
LPADDR	Same as SETC AMODE
C28ADDR	Same as CLRC AMODE
SETC MOM1MAP	Set the MOM1MAP bit in the status register.
CLRC MOM1MAP	Clear the MOM1MAP bit in the status register.
SETC CNF	Set the CNF bit (C2xlp mapping mode bit) in the status register.
CLRC CNF	Clear the CNF bit (C2xlp mapping mode bit) in the status register.
SETC XF	Set the XF bit in the status register.
CLRC XF	Clear the XF bit in the status register.

When operated in this mode, the C28x assembler generates an error if non-C27x compatible syntax or instructions are used. For example the following instructions are illegal in this mode:

```
FLIP   AL           ; C28x instruction not supported in C27x.
MOV    AL, *XAR0++  ; *XAR0++ is illegal addressing for C27x.
```

3.12.2 C28x Object Mode

This mode supports all the C28x instructions and generates C28x object code. New users of the C28x processor should use the assembler in this mode. This mode generates an error if old C27x syntax is used. For example, the following instructions are illegal in this mode:

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

3.12.3 C28x Object - Accept C27x Syntax Mode

This mode supports all the C28x instructions and also supports the C27x instruction and addressing syntax. This mode generates C28x object code. For example, this mode accepts the instruction syntax MOV AL, *AR0++ and encodes it as MOV AL, *XAR0++. Though this mode accepts C27x syntax, the assembler generates warning if C27x syntax is used to encourage the you as a programmer to change the C27x syntax to C28x syntax. The instruction MOV AL, *AR0+ generates the following warning:

```
WARNING! at line 1: [W0000] Full XAR register is modified
```

3.12.4 C28x Object - Accept C2xlp Syntax Mode

This mode supports all the C28x instructions and generates C28x object code but also supports C2xlp instruction syntax.

The C28x processor includes features and instructions that make the processor as backward compatible to the C2xlp processor as possible. In order to make the C28x processor source code compatible with C2xlp, the assembler accepts C2xlp instructions and encodes them as equivalent C28x instructions.

Refer to the TMS320C28x DSP CPU and Instruction Set Reference Guide for information on C2xlp instructions.

The C27x syntax is not supported in this mode and generates an error. Also any incompatible C2xlp instructions cause the assembler to generate an error. For example, the following instructions are illegal in this mode:

```
MOV    AL, *AR0++     ; *AR0++ is illegal addressing for C28x.
TRAP                  ; Incompatible C2XLP instruction.
```

This mode assumes LP addressing mode compatibility (AMODE = 1) and a data page of 128-words. Refer to the TMS320C28x DSP CPU and Instruction Set Reference Guide for more details.

In this mode, C28x and C2xlp source code can be freely intermixed within a file as shown below.

```
; C2xlp Source Code
        LDP     #VarA
        LACL    VarA
        LAR     AR0, *+, AR2
        SACL    *+
        .
        .
        LC      FuncA
        .
        .
; C28x Source Code using LP Addressing (AMODE = 1)
FuncA:
        MOV     DP, #VarB
        MOV     AL, @@VarB
        MOVL    XAR0, *XAR0++
        MOV     *XAR2++, AL
        LRET
```

When the C28x assembler is invoked with --asm_remarks switch, it performs additional checking for the following cases:

- The C1x/C2x/C2xx/C5x assembler accepts numbers with leading zero as decimal integers, that is 010 is treated as 10 and not as 8. The C28x assembler treats constants with leading zeros as octal numbers. There may be C2xlp assembly code that contains decimal numbers with leading zeros. When these files are assembled with the C28x assembler the results will not be what you expect as the C28x assembler treats such constants as octal numbers. So the assembler when invoked with --c2xlp_src_compatible --asm_remarks, checks for such numbers and issues a warning that the constant is parsed as an octal number.

  For example, consider the following listing produced using the --c2xlp_src_compatible --asm_remarks options:

```
    1 00000000 FF20          lacc    #023
"octal.asm", WARNING! at line 1: [W0000] Constant parsed as an octal number
      00000001 0013
```

- The C1x/C2x/C2xx/C5 assembler uses a different order of operator precedence expression. In the C1x/C2x/C2xx/C5 assembler, the shift operators (<< and >>) have higher precedence than the binary

+ and - operators. The C28x assembler follows the order of precedence of C language where the above mentioned sequence is reversed. The C28x assembler issues a warning about the precedence used if the following are true:

– The --c2xlp_src_compatible --asm_remarks options are specified.
– The source code contains any expression involving binary additive operators (+ and -) and the shift operators (<< and >>).
– The precedence is not forced by parentheses. For example, consider the following listing produced using the --c2xlp_src_compatible --asm_remarks options:

```
    1 00000000 FF20          lacc #(3 + 4 << 2)      ; Warning generated
"pre.asm", WARNING! at line 1: [W9999] The binary + and - operators have higher
precedence than the shift operators
      00000001 001C
    2 00000002 FF20          lacc #((3 + 4) << 2)    ; NO warning
      00000003 001C
```

3.12.5 C28x FPU32 Object Mode

The FPU32 mode is used when the hardware 32-bit floating-point co-processor support is available on the C28x. The FPU32 mode is invoked by specifying the -v28 and --float_support=fpu32 options. This mode supports all C28x instructions. 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 FPU32 mode assumes large memory model, and is incompatible with small memory model.
- The assembler in this mode checks for pipeline conflicts. This is because the FPU32 instructions are not pipeline protected. The 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 the FPU32 instructions: an FPU instruction can access its 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 3.18.

3.13 Source Listings

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

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

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

Field 1: Source Statement Number

Line number

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

Include file letter

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

Nesting level number

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

Field 2: Section Program Counter

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

Field 3: Object Code

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

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

Field 4: Source Statement Field

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

Figure 3-2 shows an assembler listing with each of the four fields identified.

Figure 3-2. Example Assembler Listing

```
      1               add1      .macro    S1, S2, S3,  S4
      2
      3                          MOV       AL,  S1
      4                          ADD       AL,  S2
      5                          ADD       AL,  S3
      6                          ADD       AL,  S4
      7                         .endm
      8
      9                        .global    c1, c2, c3,  c4
      10                       .global   _main
      11
      12         0001   c1       .set       1
      13         0002   c2       .set       2
      14         0003   c3       .set       3
      15         0004   c4       .set       4
      16
      17 000000         _main:
      18 000000                   add1     #c1, #c2, #c3,  #c4
1
1        000000  9A01              MOV       AL, #c1
1        000001  9C02              ADD       AL, #c2
1        000002  9C03              ADD       AL, #c3
1        000003  9C04              ADD       AL, #c4
      19
      20                         .end

     Field 1 Field 2   Field 3                Field 4
```

3.14 Debugging Assembly Source

When you invoke cl2000 -v28 with --symdebug:dwarf (or -g) when compiling an assembly file, the assembler provides symbolic debugging information that allows you to step through your assembly code in a debugger rather than using the Disassembly window in Code Composer Studio. This enables you to view source comments and other source-code annotations while debugging.

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

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

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

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

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

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

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

Example 3-5. Assembly Program for Example 3-4

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

;------------------------------------------------------------------------------
; addfive() - Add five to the second data member of _svar
;------------------------------------------------------------------------------
          .text
          .global addfive
addfive:  .asmfunc
           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
```

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

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

```
cl2000 -v28 -symdebug:dwarf cvars.c addfive.asm --run_linker --library=lnk.cmd
    --library=rts2800.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.

3.15 C-Type Symbolic Debugging for Assembly Variables (--cdebug_asm_data Option)

When you assemble with the --cdebug_asm_data option, the assembler produces the debug information for assembly source debug. The assembler outputs C-type symbolic debugging information for symbols defined in assembly source code using the data directives. This support is for basic C types, structures and arrays. You have the ability to inform the assembler how to interpret an assembly label as a C variable with basic type information.

The assembly data directives have been modified to produce debug information when using --cdebug_asm_data in these ways:

- **Data directives for initialized data**. The assembler outputs debugging information for data initialized with the .byte, .field, .float, .int, or .long directive. For the following, the assembler emits debug information to interpret init_sym as a C integer:

```
int_sym     .int   10h
```

  More than one initial value is interpreted as an array of the type designated by the directive. This example is interpreted as an integer array of four and the appropriate debug information is produced:

```
int_sym     .int   10h, 11h, 12h, 13h
```

  For symbolic information to be produced, you must have a label designated with the data directive. Compare the first and second lines of code shown below:

```
int_sym     .int   10h
            .int   11h --> Will not have debug info.
```

- **Data directives for uninitialized data**. The .bss and .usect directives accept a type designation as an optional fifth operand. This type operand is used to produce the appropriate debug information for the symbol defined using the .bss directive. For example, the following generates similar debug information as the initialized data directive shown above:

```
        .bss   int_sym,1,1,0,int
```

  The type operand can be one of the following. If a type is not specified no debug information is produced.

CHAR	FLOAT	LDOUBLE	SCHAR	UCHAR	ULONG
DOUBLE	INT	LONG	SHORT	UINT	USHORT

  In the following example, the parameter int_sym is treated as an array of four integers:

```
        .bss   int_sym,4,1,0,int
```

  The size specified must be a multiple of the type specified. If no type operand is specified no warning is issued. The following code will generate a warning since 3 is not a multiple of the size of a long.

```
        .bss   double_sym, 3,1,0,long
```

- **Debug information for assembly structures**. The assembler also outputs symbolic information on structures defined in assembly. Here is an example of a structure:

```
structlab   .struct
mem1        .int
mem2        .int
struct_len  .endstruct

struct1     .tag structlab
            .bss struct1, 2, 1, 0, structlab
```

  For the structure example, debug information is produced to treat struct1 as the C structure:

```
struct struct1{
   int mem1;
   int mem2;
     };
```

  The assembler outputs arrays of structures if the size specified by the .bss directive is a multiple of the size of struct type. As with uninitialized data directives, if the size specified is not a multiple of the structure size, a warning is generated. This example properly accounts for alignment constraints imposed by the member types:

```
        .bss struct1,struct_len * 3, 1, 0, structlab
```

3.16 Cross-Reference Listings

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

Example 3-6. 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 3-5 lists these characters and names.

Definition (DEFN) column contains the statement number that defines the symbol. This column is blank for undefined symbols.

Reference (REF) column lists the line numbers of statements that reference the symbol. A blank in this column indicates that the symbol was never used.

Table 3-5. Symbol Attributes

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

3.17 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 3-6 lists the instruction to be changed and the change that occurs.

Table 3-6. 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*

Table 3-6. Smart Encoding for Efficiency (continued)

This instruction...	Is encoded as...
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 3-7 shows a list of instructions recognized during intuitive smart encoding and what the instruction is changed to.

Table 3-7. 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 3-8 are provided; these instructions will not be optimized.

Table 3-8. 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*

3.18 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.

3.18.1 Protected and Unprotected Pipeline Instructions

The C28x target with FPU 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 FPU 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.

3.18.2 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 *TMS320C28x C/C++ Compiler User's Guide*.

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.

3.18.3 Enabling/Disabling Pipeline Conflict Detection

The PCD feature is enabled by default. To disable this feature, the compiler needs to be invoked with the --disable_pcd option. This is useful if the user wants some warnings disabled, for instance. However, care must be taken not to ignore pipeline conflict errors, as doing so might cause incorrect code execution.

3.18.4 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*.

Assembler Directives

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

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

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

Topic **Page**

4.1 Directives Summary

Table 4-1 through Table 4-10 summarize the assembler directives.

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

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

Labels and Comments Are Not Shown in Syntaxes

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

Table 4-1. Directives That Define Sections

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

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

Mnemonic and Syntax	Description	See
.byte $value_1$[**,** ...**,** $value_n$]	Initializes one or more successive bytes in the current section	.byte topic
.char $value_1$[**,** ...**,** $value_n$]	Initializes one or more successive bytes in the current section	.char topic
.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
.word $value_1$[**,** ... **,** $value_n$]	Initializes one or more 16-bit integers	.word topic
.xfloat $value_1$[**,** ...**,** $value_n$]	Places the floating-point representation of one or more floating-point 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 4-3. 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 bytes*, which must be a power of 2; defaults to 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* bits in the current section; a label points to the beginning of the reserved space	.space topic

Table 4-4. Directives That Format the Output Listing

Mnemonic and Syntax	Description	See
.drlist	Enables listing of all directive lines (default)	.drlist topic
.drnolist	Suppresses listing of certain directive lines	.drnolist topic
.fclist	Allows false conditional code block listing (default)	.fclist topic
.fcnolist	Suppresses false conditional code block listing	.fcnolist topic
.length [*page length*]	Sets the page length of the source listing	.length topic
.list	Restarts the source listing	.list topic
.mlist	Allows macro listings and loop blocks (default)	.mlist topic
.mnolist	Suppresses macro listings and loop blocks	.mnolist topic
.nolist	Stops the source listing	.nolist topic
.option $option_1$**[,** $option_2$**, . . .]**	Selects output listing options; available options are B, L, M, R, T, W, and X	.option topic
.page	Ejects a page in the source listing	.page topic
.sslist	Allows expanded substitution symbol listing	.sslist topic
.ssnolist	Suppresses expanded substitution symbol listing (default)	.ssnolist topic
.tab *size*	Sets tab to *size* characters	.tab topic
.title " *string* **"**	Prints a title in the listing page heading	.title topic
.width [*page width*]	Sets the page width of the source listing	.width topic

Table 4-5. Directives That Reference Other Files

Mnemonic and Syntax	Description	See
.copy ["]*filename*["]	Includes source statements from another file	.copy topic
.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
.include ["]*filename*["]	Includes source statements from another file	.include topic
.mlib ["]*filename*["]	Defines macro library	.mlib topic
.ref $symbol_1$**[,** ... **,** $symbol_n$]	Identifies one or more symbols used in the current module that are defined in another module	.ref topic

Table 4-6. Directives That Override the Assembly Mode

Mnemonic and Syntax	Description	See
.c28_amode	Begins assembling in C28x object mode	.c28_amode topic
.lp_amode	Begins assembling in C28x object mode -- accepts C2xLP instructions	.lp_amode topic

Table 4-7. Directives That Enable Conditional Assembly

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

Table 4-8. Directives That Define Unions or Structures

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

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

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

Table 4-10. Directives That Perform Miscellaneous Functions

Mnemonic and Syntax	Description	See
.asmfunc	Identifies the beginning of a block of code that contains a function	.asmfunc topic
.cdecls [*options*,] **"***filename***"[, "**filename2**"[,** ...]	Share C headers between C and assembly code	.cdecls topic
.clink [**"***section name***"**]	Enables conditional linking for the current or specified section	.clink topic
.emsg *string*	Sends user-defined error messages to the output device; produces no .obj file	.emsg topic
.end	Ends program	.end topic
.endasmfunc	Identifies the end of a block of code that contains a function	.endasmfunc topic
.mmsg *string*	Sends user-defined messages to the output device	.mmsg topic
.newblock	Undefines local labels	.newblock topic
.sblock	Designates section for blockins	.sblock topic
.wmsg *string*	Sends user-defined warning messages to the output device	.wmsg topic

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

This section explains how the TMS320C28x assembler directives differ from the TMS320C1x/C2x/C2xx/C5x assembler directives.

- The C28x .long and .float directives automatically align the SPC on an even word boundary, while the C1x/C2x/C2xx/C5x assembler directives do not.
- Without arguments, the .align directive for the C28x and the C1x/C2x/C2xx/C5x assemblers both align the SPC at the next page boundary. However, the C28x .align directive also accepts a constant argument, which must be a power of 2, and this argument causes alignment of the SPC on that word boundary. The .align directive for the C1x/C2x/C2xx/C5x assembler does not accept this argument.
- The .field directive for the C28x handles values of 1 to 32 bits, while the C1x/C2x/C2xx/C5x assembler handles values of 1 to 16 bits. With the C28x assembler, objects that are 16 bits or larger start on a word boundary and are placed with the least significant bits at the lower address.
- The C28x .bss and .usect directives have an additional flag called the alignment flag, which specifies alignment on an even word boundary. The C1x/C2x/C2xx/C5x .bss and .usect directives do not use this flag.
- The .string directive for the C28x initializes one character per word; the C1x/C2x/C2xx/C5x assembler directive .string, packs two characters per word. The C28x .pstring directive packs two characters per word.
- The following directives are valid with the C28x assembler but are not supported by the C1x/C2x/C2xx/C5x assembler:

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

- The .mmregs and .port directives are supported by the C1x/C2x/C2xx/C5x assembler. The C28x assembler when invoked with the --c2xlp_src_compatible option, ignores these directives and issues a warning that the directives are ignored. The C28x assembler does not accept these directives.

4.3 Directives That Define Sections

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

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

Chapter 2 discusses sections in detail.

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

The directives in Example 4-1 perform the following tasks:

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

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

Example 4-1. Sections Directives

```
                ************************************************
                *     Start assembling into the .text section     *
                ************************************************
000000                  .text
000000 0001             .word   1, 2
000001 0002
000002 0003             .word   3, 4
000003 0004

                ************************************************
                *     Start assembling into the .data section     *
                ************************************************
000000                  .data
000000 0009             .word   9, 10
000001 000A
000002 000B             .word   11, 12
000003 000C

                ************************************************
                *     Start assembling into a named,              *
                *     initialized section, var_defs               *
                ************************************************
000000                  .sect   "var_defs"
000000 0011             .word   17, 18
000001 0012

                ************************************************
                *     Resume assembling into the .data section    *
                ************************************************
000004                  .data
000004 000D             .word   13, 14
000005 000E
000000                  .bss    sym, 19    ; Reserve space in .bss
000006 000F             .word   15, 16     ; Still in .data
000007 0010

                ************************************************
                *     Resume assembling into the .text section    *
                ************************************************
000004                  .text
000004 0005             .word   5, 6
000005 0006
000000          usym    .usect  "xy", 20   ;  Reserve space in xy
000006 0007             .word   7, 8       ; Still in .text
000007 0008
```

4.4 Directives That Initialize Constants

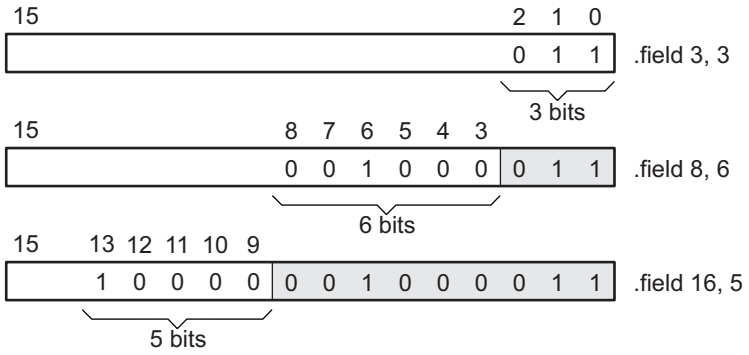
Several directives assemble values for the current section:

- The **.byte** and **.char** directives place one or more 8-bit values into consecutive words of the current section. These directives are similar to .long and .word, except that the width of each value is restricted to eight bits.
- The **.field** directive places a single value into a specified number of bits in the current word. With .field, you can pack multiple fields into a single word; the assembler does not increment the SPC until a word is filled.

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

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

Figure 4-1. The .field Directive



- The **.float** and **.xfloat** directives calculate the single-precision (32-bit) IEEE floating-point representation of a single floating-point value and store it in a word in the current section that is aligned to a word boundary.
- The **.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 .int and .word directives automatically align to a word boundary.
- The **.string** and **.pstring** directives places 8-bit characters from one or more character strings into the current section. This directive is similar to .byte, placing an 8-bit character in each consecutive byte of the current section.

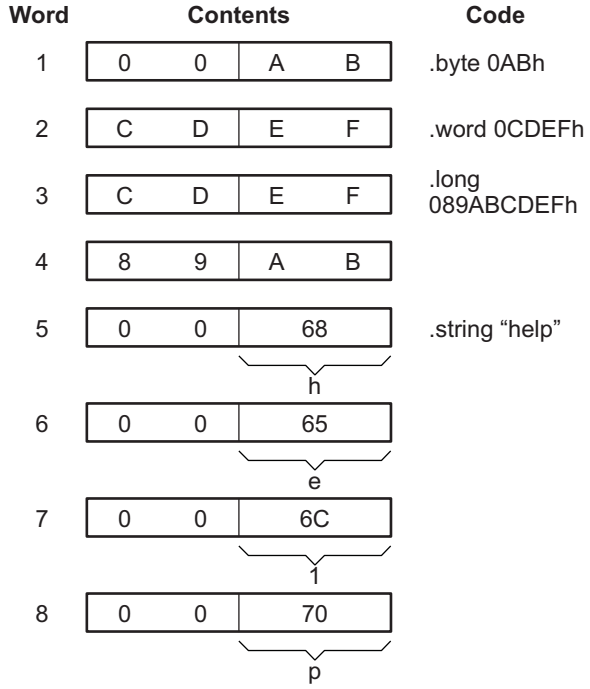
Directives That Initialize Constants When Used in a .struct/.endstruct Sequence

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

Figure 4-2 compares the .byte, .word, 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 4-2. Initialization Directives



4.5 Directives That Perform Alignment and Reserve Space

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

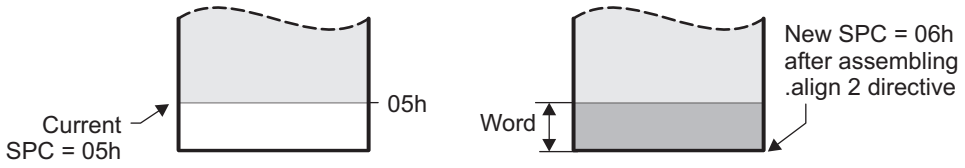
- The **.align** directive aligns the SPC at the next word boundary. This directive is useful with the .field directive when you do not want to pack two adjacent fields in the same word.

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

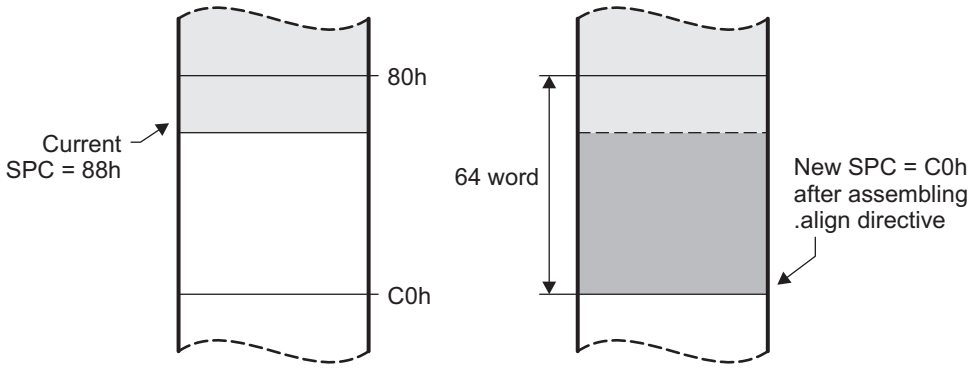
```
1 000000 0002          .field  2,3
2 000000 005A          .field  11,8
3                      .align  2
4 000002 0065          .string "errorcnt"
  000003 0072
  000004 0072
  000005 006F
  000006 0072
  000007 0063
  000008 006E
  000009 0074
5                      .align
6 000040 0004         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

Figure 4-3. The .align Directive

(a) Result of .align 2



(b) Result of .align without an argument



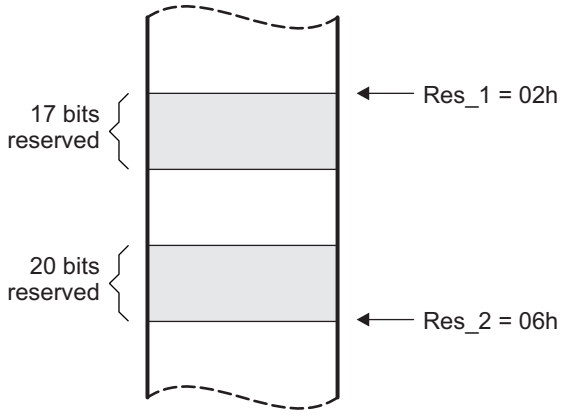
- The **.bes** and **.space** directives reserve a specified number of bits in the current section. The assembler fills these reserved bits with 0s.
  - When you use a label with .space, it points to the *first* word that contains reserved bits.
  - When you use a label with .bes, it points to the *last*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

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



4.6 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, M, T, and W).

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

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

  X produces a cross-reference listing of symbols. You can also obtain a cross-reference listing by invoking the assembler with the --cross_reference option (see Section 3.3).

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

4.7 Directives That Reference Other Files

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

- The **.copy** and **.include** directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from a copied file are printed in the listing file; the statements read from an included file are *not* printed in the listing file.
- The **.def** directive identifies a symbol that is defined in the current module and that can be used in another module. The assembler includes the symbol in the symbol table.
- The **.global** directive declares a symbol external so that it is available to other modules at link time. (For more information about global symbols, see Section 2.7.1). The .global directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. The link step 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 link step can resolve its definition. The .ref directive forces the link step to resolve a symbol reference.

4.8 Directives That Enable Conditional Assembly

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

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

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

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

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

The assembler supports several relational operators that are useful for conditional expressions. For more information about relational operators, see Section 3.9.4.

4.9 Directives That Define Unions or Structures

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

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

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

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

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

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

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

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

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

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

4.10 Directives That Define Symbols at Assembly Time

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

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

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

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

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

- The **.label** directive defines a special symbol that refers to the load-time address within the current section. This is useful when a section loads at one address but runs at a different address. For example, you may want to load a block of performance-critical code into slower off-chip memory to save space and move the code to high-speed on-chip memory to run. See the .label topic for an example using a load-time address label.
- The **.set** 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.

4.11 Directives That Override the Assembler Mode

These directives override the global syntax checking modes discussed in Section 3.12. These directives are not valid with the C27x Object Mode (-v27 option).

- The .c28_amode directive sets the assembler mode to C28x Object Mode (-v28). The instructions after this directive are assembled in C28x Object Mode regardless of the option used in the command line.
- The .lp_amode directive sets the assembler mode to C28x Object Mode - Accept C2xlp instruction syntax (--c2xlp_src_compatible). The instructions after this directives are assembled as if the --c2xlp_src_compatible options is specified on the command line.

4.12 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler -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 **.clink** directive enables conditional linking by telling the link step to leave the named section out of the final object module output of the link step if there are no references found to any symbol in the section. The .clink directive can be applied to initialized or uninitialized sections.
- The **.end** directive terminates assembly. If you use the .end directive, it should be the last source statement of a program. This directive has the same effect as an end-of-file character.
- The **.newblock** directive resets local labels. Local labels are symbols of the form $n, where n is a decimal digit, or of the form NAME?, where you specify NAME. They are defined when they appear in the label field. Local labels are temporary labels that can be used as operands for jump instructions. The .newblock directive limits the scope of local labels by resetting them after they are used. See Section 3.8.2 for information on local labels.
- The **.sblock** directive designates sections for blocking. Only initialized sections can be specified 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 5.7.

4.13 Directives Reference

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

.align — *Align SPC on the Next Boundary*

Syntax

.align [*size in 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 byte 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 link step 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/.eval *Assign a Substitution Symbol*

Syntax

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

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

Description

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

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

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

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

Example

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

```
       1                        .sslist
       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**

.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 Chapter A) to accomplish assembly debugging; those directives should be used only by the compiler to generate symbolic debugging information for C/C++ source files.

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

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

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

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

Example

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

```
 1 00000000                    .sect       ".text"
 2                             .global     userfunc
 3                             .global     _printf
 4
 5                userfunc:    .asmfunc
 6 00000000 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       ".const"
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
```

.bss *Reserve Space in the .bss Section*

Syntax

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

Description

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

- The *symbol* is a required parameter. It defines a label that points to the first location reserved by the directive. The symbol name must correspond to the variable that you are reserving space for.
- The *size in words* is a required parameter; it must be an absolute expression. The assembler allocates size words in the .bss section. There is no default size.
- The *blocking flag* is an optional parameter. If you specify a value greater than 0 for this parameter, the assembler allocates size 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 flag* is an optional parameter. It causes the assembler to allocate size in words on long word boundaries.
- The *type* is an optional parameter. Designating a *type* causes the assembler to produce the appropriate debug information for the symbol. See Section 3.15 for more information.

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

Rule 1 Whenever a hole is left in memory (as shown in Figure 4-5), the .bss directive attempts to fill it. When a .bss directive is assembled, the assembler searches its list of holes left by previous .bss directives and tries to allocate the current block into one of the holes. (This is the standard procedure regardless of whether the blocking flag has been specified.)

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

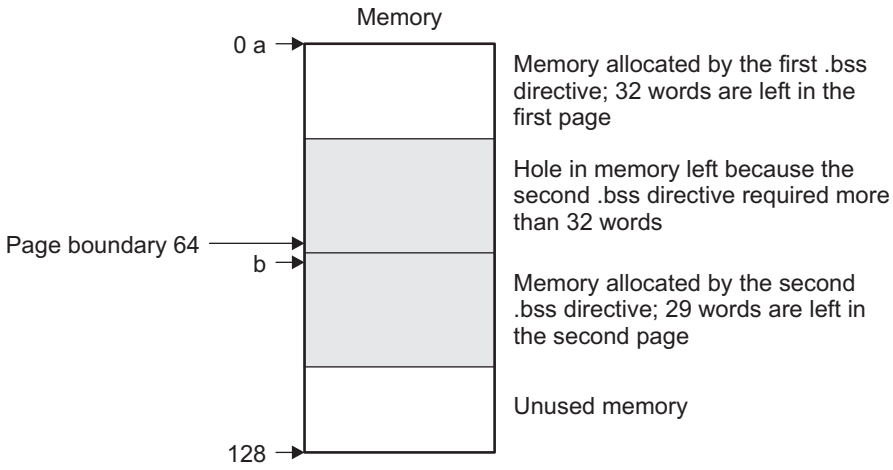
- If you do not request blocking, the memory is allocated at the current SPC.
- If you request blocking, the assembler checks to see whether there is enough space between the current SPC and the page boundary. If there is not enough space, the assembler creates another hole and allocates the space on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

The blocking option allows you to reserve up to 64 words in the .bss section and to ensure that they fit on one page of memory. (Of course, you can reserve more than 64 words at a time, but they cannot fit on a single page.) The following example code reserves two blocks of space in the .bss section.

```
memptr:    .bss     A,32,1
memptr1:   .bss     B,35,1
```

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

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



For more information about sections, see Chapter 2.

Example

In this example, the .bss directive is used to allocate space for two variables, TEMP and ARRAY. The symbol TEMP points to four words of uninitialized space (at .bss SPC = 0). The symbol ARRAY points to 100 words of uninitialized space (at .bss SPC = 040h); this space must be allocated contiguously within a page. Symbols declared with the .bss directive can be referenced in the same manner as other symbols, and they can also be declared external.

```
 1              ****************************************
 2              ** Start assembling into .text section **
 3              ****************************************
 4 000000                .text
 5 000000 2BAC           MOV     T, #0
 6
 7              ****************************************
 8              ** Allocate 4 words in .bss            **
 9              ****************************************
10 000000                .bss    Var_1, 2, 0, 1
11
12              ****************************************
13              ** Still in .text                      **
14              ****************************************
15 000001 08AC           ADD     T, #56h
   000002 0056
16 000003 3573           MPY     ACC, T, #73h
17
18
19
20
21 000040                .bss    ARRAY, 100, 1
22
23
24
25 000004 F800-          MOV     DP, #Var_1
26 000005 1E00-          MOVL    @Var_1, ACC
27              ****************************************
28              ** Declare external .bss symbol        **
29              ****************************************
30                       .global ARRAY
31                       .end
```

.byte/.char *Initialize Byte*

Syntax

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

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

Description

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

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

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. You can use up to 100 *value* parameters, but the total line length cannot exceed 200 characters.

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

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

Example

In this example, 8-bit values (10, -1, abc, and a) are placed into consecutive 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           .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
```

.c28_amode/.lp_amode *Override Assembler Mode*

Syntax

.c28_amode

.lp_amode

Description

The **.c28_amode** and **.lp_amode** directives tell the assembler to override the assembler mode. See Section 3.12 for more information on assembler modes.

The .c28_amode directive tells the assembler to operate in the C28x object mode (-v28). The .lp_amode directive tells the assembler to operate in C28x object - accept C2xlp syntax mode (--c2xlp_src_compatible). These directives can be repeated throughout a source file.

For example, if a file is assembled with the --c2xlp_src_compatible option, the assembler begins the assembly in the C28x object - accept C2xlp syntax mode. When it encounters the .c28_amode directive, it changes the mode to C28x object mode and remains in that mode until it encounters a .lp_amode directive or the end of file.

These directives help you to migrate from C2xlp to C28x by replacing a portion of the C2xlp code with C28x code.

Example

In this example, C28x code is inserted in the existing C2xlp code.

```
; C2xlp Source Code
        LDP     #VarA
        LACL    VarA
        LAR     AR0, *+, AR2
        SACL    *+
        .
        .
        CALL      FuncA
        .
        .
; The C2xlp code in function FuncA is replaced with C28x Code
; using C28x Addressing (AMODE = 0)

        .c28_amode  ; Override the assembler mode to C28x syntax
        FuncA:
        C28ADDR         ; Set AMODE to 0 C28x addressing
        MOV     DP, #VarB
        MOV     AL, @VarB
        MOVL    XAR0, *XAR0++
        MOV     *XAR2++, AL
        .lp_amode       ; Change back the assembler mode to C2xlp.
        LPADDR          ; Set AMODE to 1 to resume C2xlp addressing.
        LRET
```

.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 #define's.

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

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

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

See Chapter 12 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 link step to leave a section out of the final object module output of the link step if there are no references found to any symbol in *section name*. The .clink directive can be applied to initialized or uninitialized sections.

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

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

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

Example

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

```
 1 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 oFAB           CMPL    ACC, P
```

.copy/.include *Copy Source File*

Syntax

.copy ["]*filename*["]

.include ["]*filename*["]

Description

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

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

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

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

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

1. The directory that contains the current source file
2. Any directories named with the --include_path assembler option
3. Any directories specified by the C2000_A_DIR environment variable

For more information about the --include_path option and , see Section 3.4.

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)

```
     .space 29
     .copy "byte.asm"
** Back in original file
     .string "done"
```

byte.asm (first copy file)

```
** In byte.asm
     .byte 32,1+ 'A'
     .copy "word.asm"
** Back in byte.asm
     .byte 67h + 3q
```

word.asm (second copy file)

```
** 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. A .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.
- The *stag* is the structure's tag. Its value is associated with the beginning of the structure. If no stag is present, the assembler puts the structure members in the global symbol table with the value of their absolute offset from the top of the structure. A .stag is optional for .struct, but is required for .tag.

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 the offsets in the same manner as the C
; compiler. The resulting assembly structure can be used to access the elements of the
; C structure. Compare the difference in the offsets of those structures defined via
; .struct above and the offsets for the 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 tells the assembler to begin assembling source code into the .data section; .data becomes the current section. The .data section is normally used to contain tables of data or preinitialized variables.

For more information about sections, see Chapter 2.

Example

In this example, code is assembled into the .data 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
```

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

Syntax

.emsg *string*

.mmsg *string*

.wmsg *string*

Description

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

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

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

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

Example

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

Source file:

```
         .global     PARAM
MSG_EX   .macro parm1
         .if    $symlen(parm1) = 0
         .emsg  "ERROR -- MISSING PARAMETER"
         .else
          add   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. If any source statements follow the .end directive, the assembler ignores them.

Source file:

```
START:  .space  300
TEMP    .set    15
        .bss    LOC1, 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               .bss    LOC1, 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. If you do not specify a size, the assembler assumes the 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.

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

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

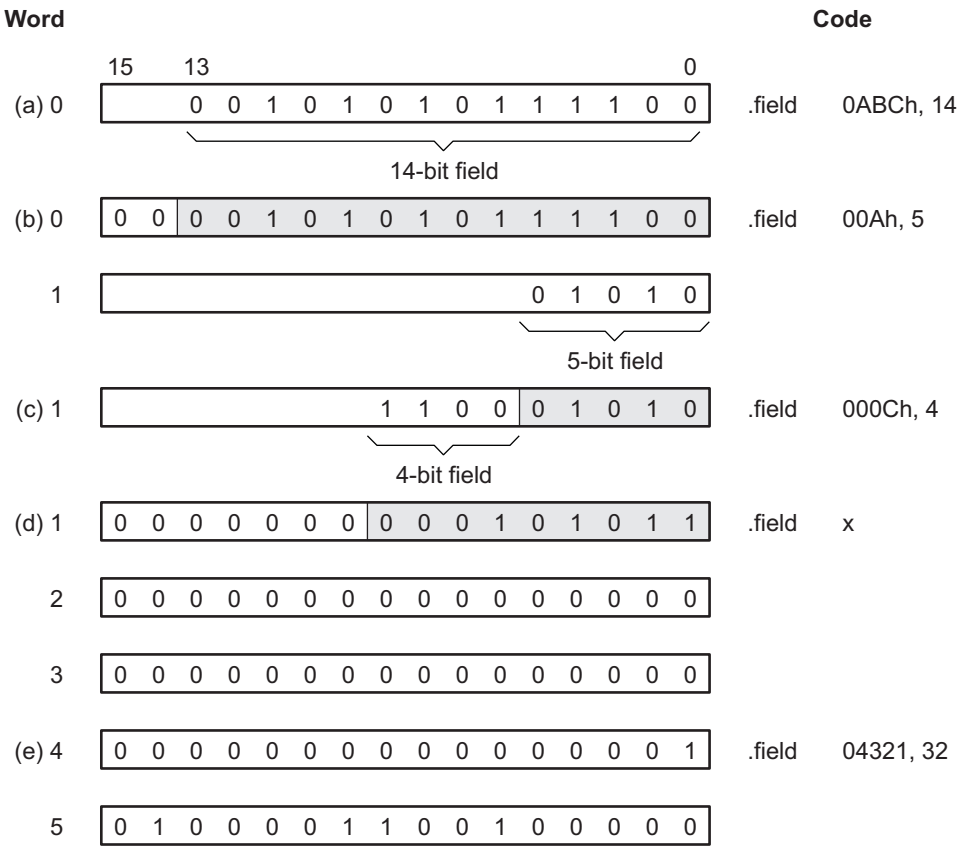
  When you use .field in a .struct/.endstruct sequence, .field defines a member's size; it does not initialize memory. For more information, see the .struct/.endstruct/.tag topic .

Example

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

```
 1                ***********************************
 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                ***********************************
19                **    22-bit relocatable field    **
20                **        in the next 2 words.    **
21                ***********************************
22 000002 0001'          .field  X
23
24                ***********************************
25                **    Initialize a 32-bit field   **
26                ***********************************
27 000003 4321           .field  04321h, 32
   000004 0000
```

Figure 4-6. The .field Directive



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

Syntax

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

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

Description

The **.float** 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 a floating-point constant or a symbol that has been equated to a floating-point constant. Each constant is converted to a floating-point value in IEEE single-precision 32-bit format.

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 4-7.

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

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

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

When you use .float 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/.globl *Identify Global Symbols*

Syntax

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

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

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

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

Description

These 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 link step resolves this symbol's definition at link time.

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

The **.globl** directive is provided for backward compatibility for C2xlp source code. It is accepted only when the --c2xlp_src_compatible option is used. The use of .globl is discouraged.

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

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

- If the symbol is *not defined in the current module* (which includes macro, copy, and include files), the .global or .ref directive tells the assembler that the symbol is defined in an external module. This prevents the assembler from issuing an unresolved reference error. At link time, the link step 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
```

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

Syntax

.if *well-defined expression*

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

[**.else**]

.endif

Description

Four directives provide conditional assembly:

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

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

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

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

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

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

See Section 3.9.4 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/.word *Initialize 16-Bit Integer*

Syntax

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

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

Description

The **.int** and **.word** 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 absolute or a relocatable expression. If an expression is relocatable, the assembler generates a relocation entry that refers to the appropriate symbol; the link step can then correctly patch (relocate) the reference. This allows you to initialize memory with pointers to variables or labels.

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

When you use .int or .word 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              .bss   PAGE, 128
3 000080              .bss   SYMPTR, 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 link step relocates it to the address at which it loads and runs.

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

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

Example

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

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

See Section 7.9 for more information about assigning run-time and load-time addresses in the link step.

.length/.width *Set Listing Page Size*

Syntax

.length [*page length*]

.width [*page width*]

Description

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

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

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

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

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

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

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

Example

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

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

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

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

Syntax

.list

.nolist

Description

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

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

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

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

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

Example

This example shows how the .copy directive inserts source statements from another file. The first time .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)

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

copy2.asm (copy file)

```
** 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/.xlong *Initialize 32-Bit Integer*

Syntax

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

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

Description

The **.long** and **.xlong** directives place one or more 32-bit values into consecutive words in the current section. The most significant word is stored first. The .long directive aligns the result on the long-word boundary, while .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 link step can then correctly patch (relocate) the reference. This allows you to initialize memory with pointers to variables or labels.

You can use up to 100 values, but they must fit on a single line (200 characters). 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 [*well-defined expression*]

.break [*well-defined expression*]

.endloop

Description

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

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

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

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

Example

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

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

.macro/.endm *Define Macro*

Syntax

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

model statements or macro directives

.endm

Description

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

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

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

Macros are explained in further detail in Chapter 5.

.mlib *Define Macro Library*

Syntax

.mlib ["]*filename*["]

Description

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

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

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

See Section 3.4 for more information about the --include_path option.

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

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

Example

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

inc1.asm	dec1.asm

`* Macro for incrementing`
`inc1 .macro A`
`ADD A, #1`
`.endm`
	`* 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 and dec1 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 5 for more information on macros and macro libraries. See the .loop/.break/.endloop topic for information on conditional blocks.

Example

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

```
       1              STR_3  .macro   P1, P2, P3
       2                     .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, cannot be used in expressions, and do not qualify for branch expansion if used with a branch. They can be used only as operands in 8-bit jump instructions. Local labels are not included in the symbol table.

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

See Section 3.8.2 for more information on the use of local labels.

Example

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

```
 1                        .ref    ADDRA, ADDRB, ADDRC
 2               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 the B, L, M, R, T, W, and X directives (turns off the limits of B, L, M, R, T, W, and X).
- **T** limits the listing of .string directives to one line.
- **W** limits the listing of .word and .int directives to one line.
- **X** produces a cross-reference listing of symbols. You can also obtain a cross-reference listing by invoking the assembler with the --cross_reference option (see Section 3.3).

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, .word, **
 3                ** .long, 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:

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

Listing file:

```
**** Page Directive Example ****                                    PAGE    1

       2            ;        .
       3            ;        .
       4            ;        .
TMS320C2000 COFF Assembler       Version x.xx          Day      Time      Year
Copyright (c) xxxx-xxxx    Texas Instruments Incorporated

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

.sblock *Specify Blocking for an Initialized Section*

Syntax

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

Description

The **.sblock** directive designates sections for blocking. Blocking is an address alignment mechanism similar to page alignment, but weaker. 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. This directive allows specification of blocking for initialized sections only, not for uninitialized sections declared with .usect or the .bss directives. The *section names* may optionally be enclosed in quotation marks.

Example

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

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

.sect *Assemble Into Named Section*

Syntax

.sect "*section name***"**

Description

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

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

See Chapter 2 for more information about sections.

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 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* word reserved.

Example

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

```
 1              ******************************************
 2              **  Begin assembling into .text section.   **
 3              ******************************************
 4  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                .bss    ADDRX, 1
       2 00000001                .bss    ADDRY, 1
       3 00000002                .bss    ADDRA, 1
       4 00000003                .bss    ADDRB, 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, ACC
#                                 MOV     @ADDRB, ACC
```

.string/.pstring *Initialize Text*

Syntax

.string {*expr*$_1$ | "*string*$_1$"}[, ... , {*expr*$_n$ | "*string*$_n$"}]

.pstring {*expr*$_1$ | "*string*$_1$"}[, ... , {*expr*$_n$ | "*string*$_n$"}]

Description

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

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. You can have up to 100 operands, but they must fit on a single source statement line.

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

When you use .string in a .struct/.endstruct sequence, .string defines a member's size; it does not initialize memory. For more information, see the .struct/.endstruct/.tag topic .

Example

In this example, 8-bit values are placed into 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. A .stag is optional for .struct, but is required for .tag.
- The *expr* is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.
- The $mem_{n/N}$ is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.
- The *element* is one of the following descriptors: .byte, .char, .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 $expr_{n/N}$ 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
           .bss REAL, 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
          .bss COMPLEX, 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
          .bss BITS, BIT_REC
```

.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** directive tells the assembler to begin assembling into the .text section, which usually contains executable code. The section program counter is set to 0 if nothing has yet been assembled into the .text section. If code has already been assembled into the .text section, the section program counter is restored to its previous value in the section.

The .text section is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you use a .data or .sect directive to specify a different section.

For more information about sections, see Chapter 2.

Example

This example assembles code into the .text and .data sections. The .data section contains integer constants and the .text section contains 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) xxxx-xxxx        Texas Instruments Incorporated

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

       2              ;        .
       3              ;        .
       4              ;        .
TMS320C2000 COFF Assembler       Version x.xx          Day      Time      Year
Copyright (c) xxxx-xxxx        Texas Instruments Incorporated

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

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

Syntax

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

Description

The **.union** directive assigns symbolic offsets to the elements of alternate data structure definitions to be allocated in the same memory space. This enables you to define several alternate structures and then let the assembler calculate the element offset. This is similar to a C union. The .union directive does not allocate any memory; it merely creates a symbolic template that can be used repeatedly.

A .struct definition can contain a .union definition, and .structs and .unions can be nested.

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

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

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

- The *utag* is the union's tag. is the union's tag. Its value is associated with the beginning of the union. If no utag is present, the assembler puts the union members in the global symbol table with the value of their absolute offset from the top of the union. In this case, each member must have a unique name.
- The *expr* is an optional expression indicating the beginning offset of the union. Unions default to start at 0. This parameter can only be used with a top-level union. It cannot be used when defining a nested union.
- The $mem_{n/N}$ is an optional label for a member of the union. This label is absolute and equates to the present offset from the beginning of the union. A label for a union member cannot be declared global.
- The *element* is one of the following descriptors: .byte, .char, .int, .long, .word, .string, .pstring, .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                 .bss  employid, 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 flag*[**,** *type*]]]

Description

The **.usect** directive reserves space for variables in an uninitialized, named section. This directive is similar to the .bss directive; both simply reserve space for data and that space has no contents. However, .usect defines additional sections that can be placed anywhere in memory, independently of the .bss section.

- The *symbol* points to the first location reserved by this invocation of the .usect directive. The symbol corresponds to the name of the variable for which you are reserving space.
- The *section name* is significant to 200 characters and must be enclosed in double quotes. This parameter names the uninitialized section. A section name can contain a subsection name in the form *section name***:***subsection name*.
- The *size in 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 object starts on a page boundary.
- The *alignment flag* is an optional parameter. It causes the assembler to allocate size in words on long word boundaries.
- The *type* is an optional parameter. Designating a *type* causes the assembler to produce the appropriate debug information for the symbol. See Section 3.15 for more information.

Initialized sections directives (.text, .data, and .sect) end the current section and tell the assembler to begin assembling into another section. A .usect or .bss directive encountered in the current section is simply assembled, and assembly continues in the current section.

Variables that can be located contiguously in memory can be defined in the same specified section; to do so, repeat the .usect directive with the same section name and the subsequent symbol (variable name).

For more information about sections, see Chapter 2.

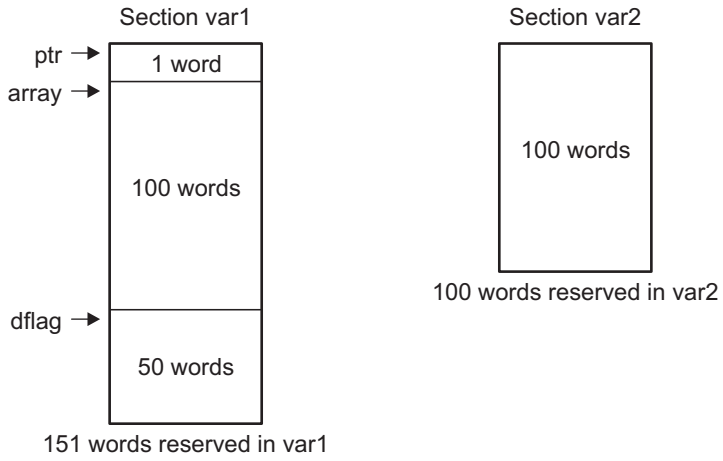
Example

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

Figure 4-8 shows how this example reserves space in two uninitialized sections, var1 and var2.

```
 1           ****************************************
 2           **    Assemble into .text section.    **
 3           ****************************************
 4 000000            .text
 5 000000 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 4-8. The .usect Directive



.var *Use Substitution Symbols as Local Variables*

Syntax

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

Description

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

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

See Chapter 5 for information on macros.

Macro 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**

5.1 Using Macros

Programs often contain routines that are executed several times. Instead of repeating the source statements for a routine, you can define the routine as a macro, then call the macro in the places where you would normally repeat the routine. This simplifies and shortens your source program.

If you want to call a macro several times but with different data each time, you can assign parameters within a macro. This enables you to pass different information to the macro each time you call it. The macro language supports a special symbol called a *substitution symbol*, which is used for macro parameters. See Section 5.3 for more information.

Using a macro is a 3-step process.

Step 1. **Define the macro.** You must define macros before you can use them in your program. There are two methods for defining macros:

a. Macros can be defined at the beginning of a *source file* or in a copy/include file. See Section 5.2, *Defining Macros*, for more information.

b. Macros can also be defined in a *macro library*. A macro library is a collection of files in archive format created by the archiver. Each member of the archive file (macro library) may contain one macro definition corresponding to the member name. You can access a macro library by using the .mlib directive. For more information, see Section 5.4.

Step 2. **Call the macro.** After you have defined a macro, call it by using the macro name as a mnemonic in the source program. This is referred to as a *macro call*.

Step 3. **Expand the macro.** The assembler expands your macros when the source program calls them. During expansion, the assembler passes arguments by variable to the macro parameters, replaces the macro call statement with the macro definition, then assembles the source code. By default, the macro expansions are printed in the listing file. You can turn off expansion listing by using the .mnolist directive. For more information, see Section 5.8.

When the assembler encounters a macro definition, it places the macro name in the opcode table. This redefines any previously defined macro, library entry, directive, or instruction mnemonic that has the same name as the macro. This allows you to expand the functions of directives and instructions, as well as to add new instructions.

5.2 Defining Macros

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file or in a .copy/.include file (see Copy Source File); they can also be defined in a macro library. For more information about macro libraries, see Section 5.4.

Macro definitions can be nested, and they can call other macros, but all elements of the macro must be defined in the same file. Nested macros are discussed in Section 5.9.

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

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

macname names the macro. You must place the name in the source statement's label field. Only the first 128 characters of a macro name are significant. The assembler places the macro name in the internal opcode table, replacing any instruction or previous macro definition with the same name.

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

parameter$_1$, *parameter*$_n$	are optional substitution symbols that appear as operands for the .macro directive. Parameters are discussed in Section 5.3.
model statements	are instructions or assembler directives that are executed each time the macro is called.
macro directives	are used to control macro expansion.
.mexit	is a directive that functions as a *goto .endm*. The .mexit directive is useful when error testing confirms that macro expansion fails and completing the rest of the macro is unnecessary.
.endm	is the directive that terminates the macro definition.

If you want to include comments with your macro definition but *do not* want those comments to appear in the macro expansion, use an exclamation point to precede your comments. If you *do* want your comments to appear in the macro expansion, use an asterisk or semicolon. See Section 5.7 for more information about macro comments.

Example 5-1 shows the definition, call, and expansion of a macro.

Example 5-1. 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
```

5.3 Macro Parameters/Substitution Symbols

If you want to call a macro several times with different data each time, you can assign parameters within the macro. The macro language supports a special symbol, called a *substitution symbol*, which is used for macro parameters.

Macro parameters are substitution symbols that represent a character string. These symbols can also be used outside of macros to equate a character string to a symbol name (see Section 3.8.6).

Valid substitution symbols can be up to 128 characters long and *must begin with a letter*. The remainder of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

Substitution symbols used as macro parameters are local to the macro they are defined in. You can define up to 32 local substitution symbols (including substitution symbols defined with the .var directive) per macro. For more information about the .var directive, see Section 5.3.6.

During macro expansion, the assembler passes arguments by variable to the macro parameters. The character-string equivalent of each argument is assigned to the corresponding parameter. Parameters without corresponding arguments are set to the null string. If the number of arguments exceeds the number of parameters, the last parameter is assigned the character-string equivalent of all remaining arguments.

If you pass a list of arguments to one parameter or if you pass a comma or semicolon to a parameter, you must surround these terms with quotation marks .

At assembly time, the assembler replaces the macro parameter/substitution symbol with its corresponding character string, then translates the source code into object code.

Example 5-2 shows the expansion of a macro with varying numbers of arguments.

Example 5-2. Calling a Macro With Varying Numbers of Arguments

Macro definition:

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

Calling the macro:

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

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

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

5.3.1 Directives That Define Substitution Symbols

You can manipulate substitution symbols with the **.asg** and **.eval** directives.

- The **.asg** directive assigns a character string to a substitution symbol.
  For the .asg directive, the quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the *substitution symbol*. The syntax of the .asg directive is:

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

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

Example 5-3. 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 5-4 shows arithmetic being performed on substitution symbols.

Example 5-4. The .eval Directive

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

In Example 5-4, the .asg directive could be replaced with the .eval directive (.eval 1, counter) without changing the output. In simple cases like this, you can use .eval and .asg interchangeably. However, you must use .eval if you want to calculate a *value* from an expression. While .asg only assigns a character string to a substitution symbol, .eval evaluates an expression and then assigns the character string equivalent to a substitution symbol.

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

5.3.2 Built-In Substitution Symbol Functions

The following built-in substitution symbol functions enable you to make decisions on the basis of the string value of substitution symbols. These functions always return a value, and they can be used in expressions. Built-in substitution symbol functions are especially useful in conditional assembly expressions. Parameters of these functions are substitution symbols or character-string constants.

In the function definitions shown in Table 5-1, *a* and *b* are parameters that represent substitution symbols or character-string constants. The term *string* refers to the string value of the parameter. The symbol *ch* represents a character constant.

Table 5-1. Substitution Symbol Functions and Return Values

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

(1) For more information about predefined register names, see Section 3.8.5.

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

Example 5-5. 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
```

5.3.3 Recursive Substitution Symbols

When the assembler encounters a substitution symbol, it attempts to substitute the corresponding character string. If that string is also a substitution symbol, the assembler performs substitution again. The assembler continues doing this until it encounters a token that is not a substitution symbol or until it encounters a substitution symbol that it has already encountered during this evaluation.

In Example 5-6, the x is substituted for z; z is substituted for y; and y is substituted for x. The assembler recognizes this as infinite recursion and ceases substitution.

Example 5-6. Recursive Substitution

```
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
```

5.3.4 Forced Substitution

In some cases, substitution symbols are not recognizable to the assembler. The forced substitution operator, which is a set of colons surrounding the symbol, enables you to force the substitution of a symbol's character string. Simply enclose a symbol with colons to force the substitution. Do not include any spaces between the colons and the symbol.

The syntax for the forced substitution operator is:

:symbol:

The assembler expands substitution symbols surrounded by colons before expanding other substitution symbols.

You can use the forced substitution operator only inside macros, and you cannot nest a forced substitution operator within another forced substitution operator.

Example 5-7 shows how the forced substitution operator is used.

Example 5-7. Using the Forced Substitution Operator

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

         .global  portbase
         force

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

5.3.5 Accessing Individual Characters of Subscripted Substitution Symbols

In a macro, you can access the individual characters (substrings) of a substitution symbol with subscripted substitution symbols. You must use the forced substitution operator for clarity.

You can access substrings in two ways:

- :symbol (*well-defined expression*):

  This method of subscripting evaluates to a character string with one character.
- :symbol (*well-defined expression*$_1$, *well-defined expression*$_2$):

  In this method, expression$_1$ represents the substring's starting position, and expression$_2$ represents the substring's length. You can specify exactly where to begin subscripting and the exact length of the resulting character string. *The index of substring characters begins with 1, not 0.*

Example 5-8 and Example 5-9 show built-in substitution symbol functions used with subscripted substitution symbols.

In Example 5-8, subscripted substitution symbols redefine the STW instruction so that it handles immediates. In Example 5-9, the subscripted substitution symbol is used to find a substring strg1 beginning at position start in the string strg2. The position of the substring strg1 is assigned to the substitution symbol pos.

Example 5-8. 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 5-9. Using Subscripted Substitution Symbols to Find Substrings

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

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

5.3.6 Substitution Symbols as Local Variables in Macros

If you want to use substitution symbols as local variables within a macro, you can use the **.var** directive to define up to 32 local macro substitution symbols (including parameters) per macro. The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

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

The .var directive is used in Example 5-8 and Example 5-9.

5.4 Macro Libraries

One way to define macros is by creating a macro library. A macro library is a collection of files that contain macro definitions. You must use the archiver to collect these files, or members, into a single file (called an archive). Each member of a macro library contains one macro definition. The files in a macro library must be unassembled source files. The macro name and the member name must be the same, and the macro filename's extension must be .asm. For example:

Macro Name	Filename in Macro Library
simple	simple.asm
add3	add3.asm

You can access the macro library by using the .mlib assembler directive (described in Define Macro Library). The syntax is:

.mlib *filename*

When the assembler encounters the .mlib directive, it opens the library named by filename and creates a table of the library's contents. The assembler enters the names of the individual members within the library into the opcode tables as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table.

The assembler expands the library entry in the same way it expands other macros. See Section 5.1 for how the assembler expands macros. You can control the listing of library entry expansions with the .mlist directive. For more information about the .mlist directive, see Section 5.8 and Start/Stop Macro Expansion Listing . Only macros that are actually called from the library are extracted, and they are extracted only once.

You can use the archiver to create a macro library by including the desired files in an archive. A macro library is no different from any other archive, except that the assembler expects the macro library to contain macro definitions. The assembler expects *only* macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable results. For information about creating a macro library archive, see Chapter 6.

5.5 Using Conditional Assembly in Macros

The conditional assembly directives are **.if/.elseif/.else/.endif** and **.loop/ .break/.endloop**. They can be nested within each other up to 32 levels deep. The format of a conditional block is:

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

The **.elseif** and **.else** directives are optional in conditional assembly. The .elseif directive can be used more than once within a conditional assembly code block. When .elseif and .else are omitted and when the .if expression is false (0), the assembler continues to the code following the .endif directive. See Assemble Conditional Blocks for more information on the .if/ .elseif/.else/.endif directives.

The **.loop/.break/.endloop** directives enable you to assemble a code block repeatedly. The format of a repeatable block is:

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

The **.loop** directive's optional *well-defined expression* evaluates to the loop count (the number of loops to be performed). If the expression is omitted, the loop count defaults to 1024 unless the assembler encounters a .break directive with an expression that is true (nonzero). See Assemble Conditional Blocks Repeatedly for more information on the .loop/.break/.endloop directives.

The **.break** directive and its expression are optional in repetitive assembly. If the expression evaluates to false, the loop continues. The assembler breaks the loop when the .break expression evaluates to true or when the .break expression is omitted. When the loop is broken, the assembler continues with the code after the .endloop directive.

For more information, see Section 4.8.

Example 5-10, Example 5-11, and Example 5-12 show the .loop/.break/ .endloop directives, properly nested conditional assembly directives, and built-in substitution symbol functions used in a conditional assembly code block.

Example 5-10. The .loop/.break/.endloop Directives

```
.asg    1,x
.loop

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

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

Example 5-11. Nested Conditional Assembly Directives

```
      .asg    1,x
      .loop

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

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

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

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

;        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
```

5.6 Using Labels in Macros

All labels in an assembly language program must be unique. This includes labels in macros. If a macro is expanded more than once, its labels are defined more than once. *Defining a label more than once is illegal*. The macro language provides a method of defining labels in macros so that the labels are unique. Simply follow each label with a question mark, and the assembler replaces the question mark with a period followed by a unique number. When the macro is expanded, *you do not see the unique number in the listing file*. Your label appears with the question mark as it did in the macro definition. You cannot declare this label as global. The syntax for a unique label is:

label **?**

Example 5-13 shows unique label generation in a macro. The maximum label length is shortened to allow for the unique suffix. For example, if the macro is expanded fewer than 10 times, the maximum label length is 126 characters. If the macro is expanded from 10 to 99 times, the maximum label length is 125. The label with its unique suffix is shown in the cross-listing file. To obtain a cross-listing file, invoke the assembler with the --cross_reference option (see Section 3.3).

Example 5-13. 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

.TMS320C2700                         000000         0
.TMS320C2800                         000001         0
.TMS320C2800_FPU32                   000000         0
.TMS320C2800_FPU64                   000000         0
__LARGE_MODEL                        000000         0
__LARGE_MODEL__                      000000         0
__TI_ASSEMBLER_VERSION_QUAL_ID__     001c52         0
__TI_ASSEMBLER_VERSION_QUAL__        000049         0
__TI_ASSEMBLER_VERSION__             4c4f28         0
__large_model__                      000000         0
l$1$                                 000004'       12     11
```

5.7 Producing Messages in Macros

The macro language supports three directives that enable you to define your own assembly-time error and warning messages. These directives are especially useful when you want to create messages specific to your needs. The last line of the listing file shows the error and warning counts. These counts alert you to problems in your code and are especially useful during debugging.

.emsg sends error messages to the listing file. The .emsg directive generates errors in the same manner as the assembler, incrementing the error count and preventing the assembler from producing an object file.

.mmsg sends assembly-time messages to the listing file. The .mmsg directive functions in the same manner as the .emsg directive but does not set the error count or prevent the creation of an object file.

.wmsg sends warning messages to the listing file. The .wmsg directive functions in the same manner as the .emsg directive, but it increments the warning count and does not prevent the generation of an object file.

Macro comments are comments that appear in the definition of the macro *but do not show up in the expansion of the macro*. An exclamation point in column 1 identifies a macro comment. If you want your comments to appear in the macro expansion, precede your comment with an asterisk or semicolon.

Example 5-14 shows user messages in macros and macro comments that do not appear in the macro expansion.

For more information about the .emsg, .mmsg, and .wmsg assembler directives, see Define Messages .

Example 5-14. 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
```

5.8 Using Directives to Format the Output Listing

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

By default, the assembler shows macro expansions and false conditional blocks in the list output file. You may want to turn this listing off or on within your listing file. Four sets of directives enable you to control the listing of this information:

- **Macro and loop expansion listing**

  .mlist expands macros and .loop/.endloop blocks. The .mlist directive prints all code encountered in those blocks.

  .mnolist suppresses the listing of macro expansions and .loop/ .endloop blocks.

  For macro and loop expansion listing, .mlist is the default.

- **False conditional block listing**

  .fclist causes the assembler to include in the listing file all conditional blocks that do not generate code (false conditional blocks). Conditional blocks appear in the listing exactly as they appear in the source code.

  .fcnolist suppresses the listing of false conditional blocks. Only the code in conditional blocks that actually assemble appears in the listing. The .if, .elseif, .else, and .endif directives do not appear in the listing.

  For false conditional block listing, .fclist is the default.

- **Substitution symbol expansion listing**

  .sslist expands substitution symbols in the listing. This is useful for debugging the expansion of substitution symbols. The expanded line appears below the actual source line.

  .ssnolist turns off substitution symbol expansion in the listing.

  For substitution symbol expansion listing, .ssnolist is the default.

- **Directive listing**

  .drlist causes the assembler to print to the listing file all directive lines.

  .drnolist suppresses the printing of certain directives in the listing file. These directives are .asg, .eval, .var, .sslist, .mlist, .fclist, .ssnolist, .mnolist, .fcnolist, .emsg, .wmsg, .mmsg, .length, .width, and .break.

  For directive listing, .drlist is the default.

5.9 Using Recursive and Nested Macros

The macro language supports recursive and nested macro calls. This means that you can call other macros in a macro definition. You can nest macros up to 32 levels deep. When you use recursive macros, you call a macro from its own definition (the macro calls itself).

When you create recursive or nested macros, you should pay close attention to the arguments that you pass to macro parameters because the assembler uses dynamic scoping for parameters. This means that the called macro uses the environment of the macro from which it was called.

Example 5-15 shows nested macros. The y in the in_block macro hides the y in the out_block macro. The x and z from the out_block macro, however, are accessible to the in_block macro.

Example 5-15. Using Nested Macros

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

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

Example 5-16 shows recursive and fact macros. The fact macro produces assembly code necessary to calculate the factorial of n, where n is an immediate value. The result is placed in the A register. The fact macro accomplishes this by calling fact1, which calls itself recursively.

Example 5-16. 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
```

5.10 Macro Directives Summary

The directives listed in Table 5-2 through Table 5-6 can be used with macros. The .macro, .mexit, .endm and .var directives are valid only with macros; the remaining directives are general assembly language directives.

Table 5-2. Creating Macros

Mnemonic and Syntax	Description	See Macro Use	See Directive
.endm	End macro definition	Section 5.2	.endm
macname **.macro** [*parameter*$_1$][**,**... **,** *parameter*$_n$]	Define macro by *macname*	Section 5.2	.macro
.mexit	Go to .endm	Section 5.2	Section 5.2
.mlib *filename*	Identify library containing macro definitions	Section 5.4	.mlib

Table 5-3. Manipulating Substitution Symbols

Mnemonic and Syntax	Description	See Macro Use	See Directive
.asg ["]*character string*["], *substitution symbol*	Assign character string to substitution symbol	Section 5.3.1	.asg
.eval *well-defined expression, substitution symbol*	Perform arithmetic on numeric substitution symbols	Section 5.3.1	.eval
.var *sym*$_1$ [**,***sym*$_2$, ...,*sym*$_n$]	Define local macro symbols	Section 5.3.6	.var

Table 5-4. Conditional Assembly

Mnemonic and Syntax	Description	See Macro Use	See Directive
.break [*well-defined expression*]	Optional repeatable block assembly	Section 5.5	.break
.endif	End conditional assembly	Section 5.5	.endif
.endloop	End repeatable block assembly	Section 5.5	.endloop
.else	Optional conditional assembly block	Section 5.5	.else
.elseif *well-defined expression*	Optional conditional assembly block	Section 5.5	.elseif
.if *well-defined expression*	Begin conditional assembly	Section 5.5	.if
.loop [*well-defined expression*]	Begin repeatable block assembly	Section 5.5	.loop

Table 5-5. Producing Assembly-Time Messages

Mnemonic and Syntax	Description	See Macro Use	See Directive
.emsg	Send error message to standard output	Section 5.7	.emsg
.mmsg	Send assembly-time message to standard output	Section 5.7	.mmsg
.wmsg	Send warning message to standard output	Section 5.7	.wmsg

Table 5-6. Formatting the Listing

Mnemonic and Syntax	Description	See Macro Use	See Directive
.fclist	Allow false conditional code block listing (default)	Section 5.8	.fclist
.fcnolist	Suppress false conditional code block listing	Section 5.8	.fcnolist
.mlist	Allow macro listings (default)	Section 5.8	.mlist
.mnolist	Suppress macro listings	Section 5.8	.mnolist
.sslist	Allow expanded substitution symbol listing	Section 5.8	.sslist
.ssnolist	Suppress expanded substitution symbol listing (default)	Section 5.8	.ssnolist

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 also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members.

Topic **Page**

6.1 Archiver Overview

You can build libraries from any type of files. Both the assembler and the linker accept archive libraries as input; the assembler can use libraries that contain individual source files, and the linker can use libraries that contain individual object files.

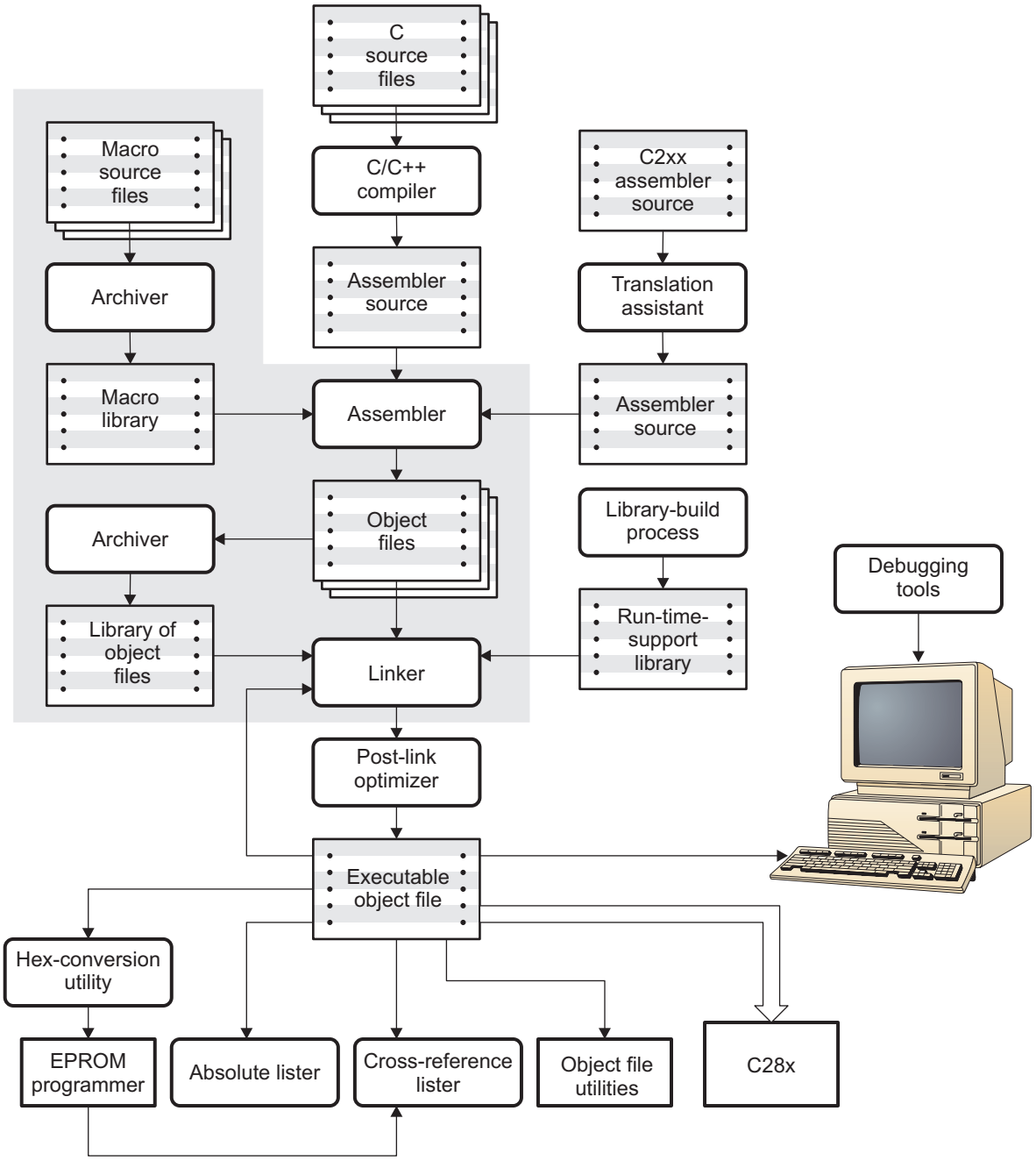
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The linker searches the library and includes members that resolve external references.

You can also use the archiver to build macro libraries. You can create several source files, each of which contains a single macro, and use the archiver to collect these macros into a single, functional group. You can use the .mlib directive during assembly to specify that macro library to be searched for the macros that you call. Chapter 5, *Macro Language*, discusses macros and macro libraries in detail, while this chapter explains how to use the archiver to build libraries.

6.2 The Archiver's Role in the Software Development Flow

Figure 6-1 shows the archiver's role in the software development process. The shaded portion highlights the most common archiver development path. Both the assembler and the linker accept libraries as input.

Figure 6-1. The Archiver in the TMS320C28x Software Development Flow



6.3 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 6-1 for an example using an archiver command file.)
- **a** adds the specified files to the library. This command does not replace an existing member that has the same name as an added file; it simply *appends* new members to the end of the archive.
- **d** deletes the specified members from the library.
- **r** replaces the specified members in the library. If you do not specify filenames, the archiver replaces the library members with files of the same name in the current directory. If the specified file is not found in the library, the archiver adds it instead of replacing it.
- **t** prints a table of contents of the library. If you specify filenames, only those files are listed. If you do not specify any filenames, the archiver lists all the members in the specified library.
- **x** extracts the specified files. If you do not specify member names, the archiver extracts all library members. When the archiver extracts a member, it simply copies the member into the current directory; it *does not* remove it from the library.

options In addition to one of the *commands,* you can specify options. To use options, combine them with a command; for example, to use the a command and the s option, enter -as or as. The hyphen is optional for archiver options only. These are the archiver options:

- **-q** (quiet) suppresses the banner and status messages.
- **-s** prints a list of the global symbols that are defined in the library. (This option is valid only with the a, r, and d commands.)
- **-u** replaces library members only if the replacement has a more recent modification date. You must use the r command with the -u option to specify which members to replace.
- **-v** (verbose) provides a file-by-file description of the creation of a new library from an old library and its members.

libname names the archive library to be built or modified. If you do not specify an extension for *libname,* the archiver uses the default extension *.lib*.

filenames names individual files to be manipulated. These files can be existing library members or new files to be added to the library. When you enter a filename, you must enter a complete filename including extension, if applicable.

Naming Library Members

Note: It is possible (but not desirable) for a library to contain several members with the same name. If you attempt to delete, replace, or extract a member whose name is the same as another library member, the archiver deletes, replaces, or extracts the first library member with that name.

6.4 Archiver Examples

The following are examples of typical archiver operations:

- If you want to create a library called function.lib that contains the files sine.obj, cos.obj, and flt.obj, enter:

```
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:

```
          FILE NAME    SIZE   DATE
----------------   -----   ------------------------
         sine.obj     300   Wed Jun 14 10:00:24 2006
          cos.obj     300   Wed Jun 14 10:00:30 2006
          flt.obj     300   Wed Jun 14 09:59:56 2006
```

- 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: '$sin'
==> symbol defined: '_cos'
==> symbol defined: '$cos'
==> symbol defined: '_tan'
==> symbol defined: '$tan'
==> symbol defined: '_atan
==> symbol defined: '$atan'
==> building archive 'function.lib'
```

  Because this example does not specify an extension for the libname, the archiver adds the files to the library called function.lib. If function.lib does not exist, the archiver creates it. (The -s option tells the archiver to list the global symbols that are defined in the library.)

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

```
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 6-1 is the modules.cmd command file. The r command specifies that the filenames given in the command file replace files of the same name in the modules.lib library. The -u option specifies that these files are replaced only when the current file has a more recent revision date than the file that is in the library.

Example 6-1. Archiver Command File

```
; Command file to replace members of the
;     modules library with updated files
; Use r command and u option:
ru
; Specify library name:
modules.lib
; List filenames to be replaced if updated:
align.obj
bss.obj
data.obj
text.obj
sect.obj
clink.obj
copy.obj
double.obj
drnolist.obj
emsg.obj
end.obj
```

Link Step Description

The TMS320C28x™ link step creates executable modules by combining object modules. This chapter describes the link step 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 link step operation; Chapter 2 discusses the object module sections in detail.

Topic **Page**

7.1 Link Step Overview

The TMS320C28x link step allows you to configure system memory by allocating output sections efficiently into the memory map. As the link step 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

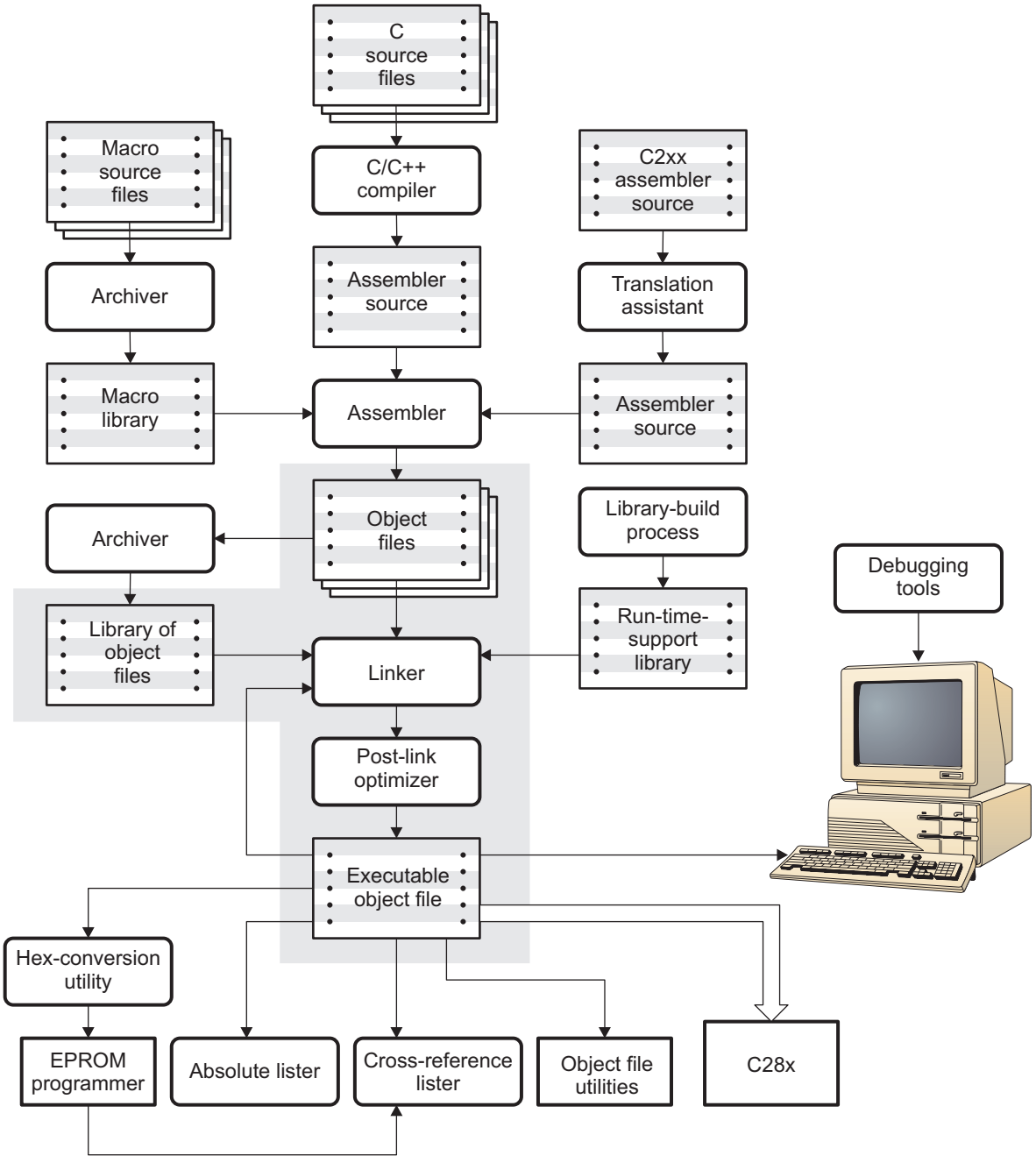
The link step command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS,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

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

7.2 The Link Step's Role in the Software Development Flow

Figure 7-1 illustrates the link step's role in the software development process. The link step accepts several types of files as input, including object files, command files, libraries, and partially linked files. The link step creates an executable object module that can be downloaded to one of several development tools or executed by a TMS320C28x device.

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



7.3 Invoking the Link Step

The general syntax for invoking the link step is:

cl2000 -v28 --run_linker [*options*] *filename*$_1$ .... *filename*$_n$

cl2000 -v28 --run_linker	is the command that invokes the link step.
options	can appear anywhere on the command line or in a link command file. (Options are discussed in Section 7.4.)
filename$_1$, *filename*$_n$	can be object files, link command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The link step can determine whether the input file is an object or ASCII file that contains link step commands. The default output filename is *a.out*, unless you use the --output_file option to name the output file.

There are two methods for invoking the link step:

- Specify options and filenames on the command line. This example links two files, file1.obj and file2.obj, and creates an output module named link.out.

  ```
  cl2000 -v28 --run_linker file1.obj file2.obj --output_file=link.out
  ```

- Put filenames and options in a link command file. Filenames that are specified inside a link command file must begin with a letter. For example, assume the file linker.cmd contains the following lines:

  ```
  --output_file=link.out
  file1.obj
  file2.obj
  ```

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

  ```
  cl2000 -v28 --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 -v28 --run_linker --map_file=link.map linker.cmd file3.obj
  ```

  The link step reads and processes a command file as soon as it encounters the filename on the command line, so it links the files in this order: file1.obj, file2.obj, and file3.obj. This example creates an output file called link.out and a map file called link.map.

For information on invoking the link step for C/C++ files, see Section 7.18.

7.4 Link Step Options

Link step options control linking operations. They can be placed on the command line or in a command file. Link step options must be preceded by a hyphen (-). Options can be separated from arguments (if they have them) by an optional space. Table 7-1 summarizes the link step options.

Table 7-1. Link Step 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 link step acts as if --absolute_exe were specified.	Section 7.4.1.1
-ar		Produces a relocatable, executable object module	Section 7.4.1.3
--arg_size	--args	Allocates memory to be used by the loader to pass arguments	Section 7.4.2
--disable_clink	-j	Disables conditional linking of COFF object modules	Section 7.4.3
--entry_point	-e	Defines a global symbol that specifies the primary entry point for the output module	Section 7.4.4
--fill_value	-f	Sets default fill values for holes within output sections; *fill_value* is a 32-bit constant	Section 7.4.5
--gen_func_subsections		Puts each function in its own subsection in the object file	
--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 bytes	Section 7.4.6
--library	-l	Names an archive library or link command *filename* as link step input	Section 7.4.7
--linker_help	-help	Displays information about syntax and available options	–
--make_global	-g	Makes *symbol* global (overrides -h)	Section 7.4.8
--make_static	-h	Makes all global symbols static	Section 7.4.9
--map_file	-m	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename*	Section 7.4.10
--no_sym_merge	-b	Disables merge of symbolic debugging information in COFF object files	Section 7.4.11
--no_sym_table	-s	Strips symbol table information and line number entries from the output module	Section 7.4.12
--output_file	-o	Names the executable output module. The default filename is a.out.	Section 7.4.13
--priority	-priority	Satisfies unresolved references by the first library that contains a definition for that symbol	Section 7.4.15
--ram_model	-cr	Initializes variables at load time	Section 7.4.14
--relocatable	-r	Produces a nonexecutable, relocatable output module	Section 7.4.1.2
--reread_libs	-x	Forces rereading of libraries, which resolves back references	Section 7.4.15
--rom_model	-c	Autoinitializes variables at run time	Section 7.4.14
--run_abs	-abs	Produces an absolute listing file	Section 7.4.16
--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 7.4.7.1
--stack_size	-stack	Sets C system stack size to *size* words and defines a global symbol that specifies the stack size. Default = 1K bytes	Section 7.4.17
--undef_sym	-u	Places an unresolved external *symbol* into the output module's symbol table	Section 7.4.19
--warn_sections	-w	Displays a message when an undefined output section is created	Section 7.4.20
--xml_link_info		Generates a well-formed XML *file* containing detailed information about the result of a link	Section 7.4.21

7.4.1 Relocation Capabilities (--absolute_exe and --relocatable Options)

The link step performs relocation, which is the process of adjusting all references to a symbol when the symbol's address changes. The link step supports two options (--absolute_exe and --relocatable) that allow you to produce an absolute or a relocatable output module. The link step also supports a third option (-ar) that allows you to produce an executable, relocatable output module.

When the link step 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 link step created it).

7.4.1.1 Producing an Absolute Output Module (--absolute_exe option)

When you use the --absolute_exe option without the --relocatable option, the link step produces an *absolute, executable* output module. Absolute files contain *no* relocation information. Executable files contain the following:

- Special symbols defined by the link step (see Section 7.14.4)
- An optional header that describes information such as the program entry point
- *No* unresolved references

The following example links file1.obj and file2.obj and creates an absolute output module called a.out:

```
cl2000 -v28 --run_linker --absolute_exe file1.obj file2.obj
```

The --absolute_exe and --relocatable Options

Note: If you do not use the --absolute_exe or the --relocatable option, the link step acts as if you specified --absolute_exe.

7.4.1.2 Producing a Relocatable Output Module (--relocatable option)

When you use the -ar option, the link step retains relocation entries in the output module. If the output module is relocated (at load time) or relinked (by another link step execution), use --relocatable to retain the relocation entries.

The link step 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 link step symbols or an optional header. The file can contain unresolved references, but these references do not prevent creation of an output module.

This example links file1.obj and file2.obj and creates a relocatable output module called a.out:

```
cl2000 -v28 --run_linker --relocatable file1.obj file2.obj
```

The output file a.out can be relinked with other object files or relocated at load time. (Linking a file that will be relinked with other files is called partial linking. For more information, see Section 7.17.)

7.4.1.3 Producing an Executable, Relocatable Output Module (-ar Option)

If you invoke the link step with both the --absolute_exe and --relocatable options, the link step produces an *executable, relocatable* object module. The output file contains the special link step symbols, an optional header, and all resolved symbol references; however, the relocation information is retained.

This example links file1.obj and file2.obj and creates an executable, relocatable output module called xr.out:

```
cl2000 -v28 --run_linker -ar file1.obj file2.obj --output_file=xr.out
```

7.4.2 Allocate Memory for Use by the Loader to Pass Arguments (--arg_size Option)

The --arg_size option instructs the link step to allocate memory to be used by the loader to pass arguments from the command line of the loader to the program. The syntax of the --arg_size option is:

--arg_size= *size*

The *size* is a number representing the number of bytes to be allocated in target memory for command-line arguments.

By default, the link step creates the __c_args__ symbol and sets it to -1. When you specify --arg_size=*size*, the following occur:

- The link step 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 C/C++ Compiler User's Guide* for information about the loader.

7.4.3 Disable Conditional Linking (--disable_clink option)

The --disable_clink option disables removal of unreferenced sections in COFF object modules. Only sections marked as candidates for removal with the .clink assembler directive are affected by conditional linking. See Conditionally Leave Section Out of Object Module Output for details on setting up conditional linking using the .clink directive.

7.4.4 Define an Entry Point (--entry_point=global_symbol Option)

The memory address at which a program begins executing is called the *entry point*. When a loader loads a program into target memory , the program counter (PC) must be initialized to the entry point; the PC then points to the beginning of the program.

The link step can assign one of four values to the entry point. These values are listed below in the order in which the link step 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 as an external symbol of the input files.
- The value of symbol _c_int00 (if present). The _c_int00 symbol *must* be the entry point if you are linking code produced by the C compiler.
- The value of symbol _main (if present)
- 0 (default value)

This example links file1.obj and file2.obj. The symbol begin is the entry point; begin must be defined as external in file1 or file2.

```
cl2000 -v28 --run_linker --entry_point=begin file1.obj file2.obj
```

7.4.5 Set Default Fill Value (--fill_value=value Option)

The --fill_value option fills the holes formed within output sections. The syntax for the --fill_value 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 link step uses 0 as the default fill value.

This example fills holes with the hexadecimal value ABCDABCD:

```
cl2000 -v28 --run_linker --fill_value=0xABCDABCD file1.obj file2.obj
```

7.4.6 Define Heap Size (--heap_size= size Option)

The C/C++ compiler uses an uninitialized section called .sysmem for the C run-time memory pool used by malloc(). You can set the size of this memory pool at link time by using the --heap_size option. The syntax for the --heap_size option is:

--heap_size= *size*

The *size* must be a constant. This example defines a 4K byte heap:

```
cl2000 -v28 --run_linker --heap_size=0x1000 /* defines a 4k heap (.sysmem section)*/
```

The link step creates the .sysmem section only if there is a .sysmem section in an input file.

The link step also creates a global symbol __SYSMEM_SIZE and assigns it a value equal to the size of the heap. The default size is 1K words.

For more information about C/c++ linking, see Section 7.18.

7.4.7 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 link step input, you simply enter the filename; the link step looks for the file in the current directory. For example, suppose the current directory contains the library object.lib. Assume that this library defines symbols that are referenced in the file file1.obj. This is how you link the files:

```
cl2000 -v28 --run_linker file1.obj object.lib
```

If you want to use a file that is not in the current directory, use the --library link step option. The syntax for this option is:

--library=[*pathname*] *filename*

The *filename* is the name of an archive, an object file, or link command file. You can specify up to 128 search paths.

The --library option is not required when one or more members of an object library are specified for input to an output section. For more information about allocating archive members, see Section 7.8.7.

You can augment the link step's directory search algorithm by using the --search_path link step option or the C2000_C_DIR environment variable. The link step searches for object libraries and command files in the following order:

1. It searches directories named with the --search_path link step 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.

7.4.7.1 Name an Alternate Library Directory (--search_path=pathname Option)

The --search_path option names an alternate directory that contains input files. The syntax for this option is:

--search_path=*pathname*

The *pathname* names a directory that contains input files.

When the link step 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. The table below shows the directories that r.lib and lib2.lib reside in, how to set environment variable, and how to use both libraries during a link. Select the row for your operating system:

Operating System	Pathname	Enter
UNIX (Bourne shell)	/ld and /ld2	`cl2000 -v28 --run_linker f1.obj f2.obj --search_path=/ld --search_path=/ld2 --library=r.lib --library=lib2.lib`
Windows	\ld and \ld2	`cl2000 -v28 --run_linker f1.obj f2.obj --search_path=\ld --search_path=\ld2 --library=r.lib --library=lib2.lib`

7.4.7.2 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 link step 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 link step option on the command line or in a command file to tell the link step which library or link command file to search for. The pathnames must follow these constraints:

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

  ```
  set 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 the directories that r.lib and lib2.lib reside in, how to set the environment variable, and how to use both libraries during a link. Select the row for your operating system:

Operating System	Pathname	Invocation Command
UNIX (Bourne shell)	/ld and /ld2	`C2000_C_DIR="/ld ;/ld2"; export C2000_C_DIR; cl2000 -v28 --run_linker f1.obj f2.obj --library=r.lib --library=lib2.lib`
Windows	\ld and \ld2	`set C2000_C_DIR=\ld;\ld2` `cl2000 -v28 --run_linker f1.obj f2.obj --library=r.lib --library=lib2.lib`

The environment variable remains set until you reboot the system or reset the variable by entering:

Operating System	Enter
UNIX (Bourne shell)	`unset 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 link step searches for object libraries in the directories named with C2000_A_DIR. For more information about object libraries, see Section 7.6.

7.4.8 Make a Symbol Global (--make_global=symbol Option)

The --make_static option makes all global symbols static. If you have a symbol that you want to remain global and you use the --make_static option, you can use the --make_global option to declare that symbol to be global. The --make_global option overrides the effect of the --make_static option for the symbol that you specify. The syntax for the --make_global option is:

--make_global= *global_symbol*

7.4.9 Make All Global Symbols Static (--make_static Option)

The --make_static option makes all global symbols static. Static symbols are not visible to externally linked modules. By making global symbols static, global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.

The --make_static option effectively nullifies all .global assembler directives. All symbols become local to the module in which they are defined, so no external references are possible. For example, assume file1.obj and file2.obj both define global symbols called EXT. By using the --make_static option, you can link these files without conflict. The symbol EXT defined in file1.obj is treated separately from the symbol EXT defined in file2.obj.

```
cl2000 -v28 --run_linker --make_static file1.obj file2.obj
```

7.4.10 Create a Map File (--map_file=filename Option)

The --map_file option creates a link step map listing and puts it in *filename.* The syntax for the --map_file option is:

--map_file= *filename*

The link step map describes:

- Memory configuration
- Input and output section allocation
- The addresses of external symbols after they have been relocated

The map file contains the name of the output module and the entry point; it can also contain up to three tables:

- A table showing the new memory configuration if any nondefault memory is specified (memory configuration). The table has the following columns; this information is generated on the basis of the information in the MEMORY directive in the link command file:
  - **Name.** This is the name of the memory range specified with the MEMORY directive.
  - **Origin.** This specifies the starting address of a memory range.
  - **Length.** This specifies the length of a memory range.
  - **Unused.** This specifies the total amount of unused (available) memory in that memory area.
  - **Attributes.** This specifies one to four attributes associated with the named range:

    R specifies that the memory can be read.

    W specifies that the memory can be written to.

    X specifies that the memory can contain executable code.

    I specifies that the memory can be initialized.

- For more information about the MEMORY directive, see Section 7.7.
- A table showing the linked addresses of each output section and the input sections that make up the output sections (section allocation map). This table has the following columns; this information is generated on the basis of the information in the SECTIONS directive in the link command file:
  - **Output section.** This is the name of the output section specified with the SECTIONS directive.
  - **Origin.** The first origin listed for each output section is the starting address of that output section. The indented origin value is the starting address of that portion of the output section.
  - **Length.** The first length listed for each output section is the length of that output section. The indented length value is the length of that portion of the output section.
  - **Attributes/input sections.** This lists the input file or value associated with an output section. If the input section could not be allocated, the map file will indicate this with "FAILED TO ALLOCATE".
- For more information about the SECTIONS directive, see Section 7.8.
- 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.

This following example links file1.obj and file2.obj and creates a map file called map.out:

```
cl2000 -v28 --run_linker file1.obj file2.obj --map_file=map.out
```

Example 7-27 shows an example of a map file.

7.4.11 Disable Merge of Symbolic Debugging Information (--no_sym_merge Option)

By default, the link step eliminates duplicate entries of symbolic debugging information. Such duplicate information is commonly generated when a C program is compiled for debugging. For example:

```
-[ header.h ]-
typedef struct
{
   <define some structure members>
} XYZ;

-[ f1.c ]-
#include "header.h"
...

-[ f2.c ]-
#include "header.h"
...
```

When these files are compiled for debugging, both f1.obj and f2.obj have symbolic debugging entries to describe type XYZ. For the final output file, only one set of these entries is necessary. The link step eliminates the duplicate entries automatically.

Use the COFF only --no_sym_merge option if you want the link step to keep such duplicate entries in COFF object files. Using the --no_sym_merge option has the effect of the link step running faster and using less machine memory.

7.4.12 Strip Symbolic Information (--no_sym_table Option)

The --no_sym_table option creates a smaller output module by omitting symbol table information and line number entries. The --no_sym_table option is useful for production applications when you do not want to disclose symbolic information to the consumer.

This example links file1.obj and file2.obj and creates an output module, stripped of line numbers and symbol table information, named nosym.out:

```
cl2000 -v28 --run_linker --output_file=nosym.out --no_sym_table file1.obj file2.obj
```

Using the --no_sym_table option limits later use of a symbolic debugger.

Stripping Symbolic Information

Note: To remove symbol table information, use the strip2000 utility as described in Section 10.3. The --no_sym_table option is deprecated.

7.4.13 Name an Output Module (--output_file=filename Option)

The link step creates an output module when no errors are encountered. If you do not specify a filename for the output module, the link step gives it the default name a.out. If you want to write the output module to a different file, use the --output_file option. The syntax for the --output_file option is:

--output_file= *filename*

The *filename* is the new output module name.

This example links file1.obj and file2.obj and creates an output module named run.out:

```
cl2000 -v28 --run_linker --output_file=run.out file1.obj file2.obj
```

7.4.14 C Language Options (--ram_model and --rom_model Options)

The --ram_model and --rom_model options cause the link step to use linking conventions that are required by the C compiler.

- The --ram_model option tells the link step to initialize variables at load time.
- The --rom_model option tells the link step to autoinitialize variables at run time.

For more information, see Section 7.18, Section 7.18.4, and Section 7.18.5.

7.4.15 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 link step 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 link step to reread all libraries. The link step 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 -v28 --run_linker --library=a.lib --library=b.lib --library=a.lib
```

or you can force the link step to do it for you:

```
cl2000 -v28 --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 -v28 --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.lib without providing a full replacement for rts2800.lib. Using --priority and linking your new library before rts2800.lib guarantees that all references to malloc and free resolve to the new library.

The --priority option is intended to support linking programs with DSP/BIOS where situations like the one illustrated above occur.

7.4.16 Create an Absolute Listing File (--run_abs Option)

The --run_abs option produces an output file for each file that was linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

7.4.17 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 bytes. This example defines a 4K byte stack:

```
cl2000 -v28 --run_linker --stack_size=0x1000 /* defines a 4K stack (.stack section) */
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size is different.

When the link step defines the .stack section, it also defines a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the section. The default software stack size is 1K words.

7.4.18 Mapping of Symbols (--symbol_map Option)

Symbol mapping allows a symbol reference to be resolved by a symbol with 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 link step resolve any references to foo by the definition foo_patch:

```
--symbol_map='foo=foo_patch'
```

7.4.19 Introduce an Unresolved Symbol (--undef_sym=symbol Option)

The --undef_sym option introduces an unresolved symbol into the link step's symbol table. This forces the link step to search a library and include the member that defines the symbol. The link step 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.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 link step to search rts2800.lib for the member that defines symtab and to link in the member.

```
cl2000 -v28 --run_linker --undef_sym=symtab file1.obj file2.obj rts2800.lib
```

If you do not use --undef_sym, this member is not included, because there is no explicit reference to it in file1.obj or file2.obj.

7.4.20 Display a Message When an Undefined Output Section Is Created (--warn_sections Option)

In a link command file, you can set up a SECTIONS directive that describes how input sections are combined into output sections. However, if the link step encounters one or more input sections that do not have a corresponding output section defined in the SECTIONS directive, the link step combines the input sections that have the same name into an output section with that name. By default, the link step does not display a message to tell you that this occurred.

You can use the --warn_sections option to cause the link step to display a message when it creates a new output section.

For more information about the SECTIONS directive, see Section 7.8. For more information about the default actions of the link step, see Section 7.13.

7.4.21 Generate XML Link Information File (--xml_link_info Option)

The link step supports the generation of an XML link information file through the --xml_link_info=*file* option. This option causes the link step 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 link step generated map file.

See Chapter B for specifics on the contents of the generated XML file.

7.5 Link Step Command Files

Link step command files allow you to put linking information in a file; this is useful when you invoke the link step often with the same information. Link step 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.

Link step 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 link step does not return from called command files.)
- Link step 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 link step directives. The MEMORY directive defines the target memory configuration (see Section 7.7). The SECTIONS directive controls how sections are built and allocated (see Section 7.8.)
- Assignment statements, which define and assign values to global symbols

To invoke the link step with a command file, enter the cl2000 -v28 --run_linker command and follow it with the name of the command file:

cl2000 -v28 --run_linker *command_filename*

The link step processes input files in the order that it encounters them. If the link step recognizes a file as an object file, it links the file. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it. Command filenames are case sensitive, regardless of the system used.

Example 7-1 shows a sample link command file called link.cmd.

Example 7-1. Link Step Command File

```
a.obj                   /*  First input filename          */
b.obj                   /*  Second input filename         */
--output_file=prog.out   /*  Option to specify output file */
--map_file=prog.map      /*  Option to specify map file    */
```

The sample file in Example 7-1 contains only filenames and options. (You can place comments in a command file by delimiting them with /* and */.) To invoke the link step with this command file, enter:

```
cl2000 -v28 --run_linker link.cmd
```

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

```
cl2000 -v28 --run_linker --relocatable link.cmd c.obj d.obj
```

The link step processes the command file as soon as it encounters the filename, so a.obj and b.obj are linked into the output module before c.obj and d.obj.

You can specify multiple command files. If, for example, you have a file called names.lst that contains filenames and another file called dir.cmd that contains link step directives, you could enter:

```
cl2000 -v28 --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 link step directives in a command file. Example 7-2 shows a sample command file that contains link step directives.

Example 7-2. Command File With Link Step 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
  .bss:   > FAST_MEM
}
```

For more information, see Section 7.7 for the MEMORY directive, and Section 7.8 for the SECTIONS directive.

7.5.1 Reserved Names in Link Step Command Files

The following names are reserved as keywords for link step directives. Do not use them as symbol or section names in a command file.

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

7.5.2 Constants in Link Step Command Files

You can specify constants with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants used in the assembler (see Section 3.6) or the scheme used for integer constants in C syntax.

Examples:

Format	Decimal	Octal	Hexadecimal
Assembler format	32	40q	020h
C format	32	040	0x20

7.6 Object Libraries

An object library is a partitioned archive file that contains object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as link step input, the link step includes any members of the library that define existing unresolved symbol references. You can use the archiver to build and maintain libraries. Chapter 6 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 link step 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 7.4.15). A library has a table that lists all external symbols defined in the library; the link step searches through the table until it determines that it cannot use the library to resolve any more references.

The following examples link several files and libraries, using these assumptions:

- Input files f1.obj and f2.obj both reference an external function named *clrscr*.
- Input file f1.obj references the symbol *origin*.
- Input file f2.obj references the symbol *fillclr*.
- Member 0 of library libc.lib contains a definition of *origin*.
- Member 3 of library liba.lib contains a definition of *fillclr*.
- Member 1 of both libraries defines *clrscr*.

If you enter:

```
cl2000 -v28 --run_linker f1.obj f2.obj liba.lib libc.lib
```

then:

- Member 1 of liba.lib satisfies the f1.obj and f2.obj references to *clrscr* because the library is searched and the definition of *clrscr* is found.
- Member 0 of libc.lib satisfies the reference to *origin*.
- Member 3 of liba.lib satisfies the reference to *fillclr*.

If, however, you enter:

```
cl2000 -v28 --run_linker fl.obj f2.obj libc.lib liba.lib
```

then the references to *clrscr* are satisfied by member 1 of libc.lib.

If none of the linked files reference symbols defined in a library, you can use the --undef_sym option to force the link step to include a library member. (See Section 7.4.19.) The next example creates an undefined symbol rout1 in the link step's global symbol table:

```
cl2000 -v28 --run_linker --undef_sym=rout1 libc.lib
```

If any member of libc.lib defines rout1, the link step includes that member.

Library members are allocated according to the SECTIONS directive default allocation algorithm; see Section 7.8.

Section 7.4.7 describes methods for specifying directories that contain object libraries.

7.7 The MEMORY Directive

The link step 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 link step 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 2.3 and Section 2.4.

7.7.1 Default Memory Model

If you do not use the MEMORY directive, the link step uses a default memory model that is based on the TMS320C28x architecture. For more information about the default memory model, see Section 7.13.

7.7.2 MEMORY Directive Syntax

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

- Name
- Starting address
- Length
- Optional set of attributes
- Optional fill specification

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 7.11.)

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.)

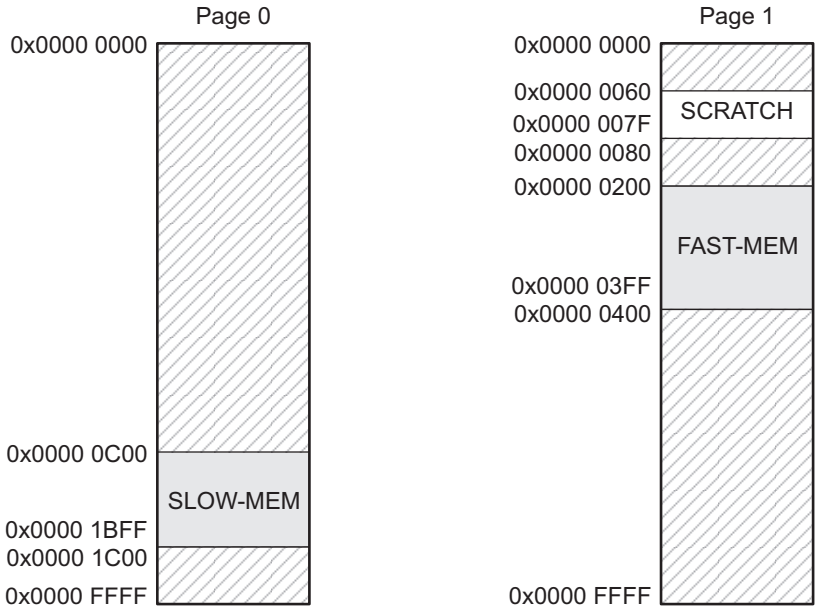
When you use the MEMORY directive, be sure to identify all memory ranges that are available for loading code. Memory defined by the MEMORY directive is configured; any memory that you do not explicitly account for with MEMORY is unconfigured. The link step does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in Example 7-3 defines a system that has 4K 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.

Example 7-3. The MEMORY Directive

```
/*********************************************************/
/*         Sample command file with MEMORY directive     */
/*********************************************************/
file1.obj    file2.obj                    /*  Input files  */
--output_file=prog.out                    /*  Options      */

MEMORY
{
   PAGE 0:  SLOW_MEM: origin = 0x00000C00, length = 0x00001000

   PAGE 1:  SCRATCH:  origin = 0x00000060, length = 0x00000020
            FAST_MEM: origin = 0x00000200, length = 0x00000200
}
```

Figure 7-2. Memory Map Defined in Example 7-3



The general syntax for the MEMORY directive is:

MEMORY

{

[**PAGE 0:**] *name 1* [**(***attr***)**] **: origin =** *constant*, **length =** *constant* [**, fill =** *constant*];

[**PAGE 1:**] *name 2* [**(***attr***)**] **: origin =** *constant*, **length =** *constant* [**, fill =** *constant*];

.

.

[**PAGE** *n*:] *name n* [**(***attr***)**] **: origin =** *constant*, **length =** *constant* [**, 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. If you do not specify PAGE, the linker uses PAGE 0. Each PAGE represents a completely independent address space. Configured memory on PAGE 0 can overlap configured memory on PAGE 1 and so on.

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 link step; they simply identify memory ranges. Memory range names are internal to the link step 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 22-bit constant and 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 constant and can be decimal, octal, or hexadecimal.

fill specifies a fill character for the memory range; enter as *fill* or *f*. Fills are optional. The value is a integer constant and can be decimal, octal, or hexadecimal. The fill value is used to fill areas of the memory range that are not allocated to a section.

Filling Memory Ranges

Note: If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFFFFFh:

```
MEMORY
{
    RFILE (RW) : o = 0x00000020, l = 0x00001000, f = 0xFFFFFFFFh
}
```

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control allocation of output sections. After you use MEMORY to specify the target system's memory model, you can use SECTIONS to allocate output sections into specific named memory ranges or into memory that has specific attributes. For example, you could allocate the .text and .data sections into the area named FAST_MEM and allocate the .bss section into the area named SLOW_MEM.

7.8 The SECTIONS Directive

The SECTIONS directive:

- Describes how input sections are combined into output sections
- Defines output sections in the executable program
- Specifies where output sections are placed in memory (in relation to each other and to the entire memory space)
- Permits renaming of output sections

For more information, see Section 2.3, Section 2.4, and Section 2.2.4. Subsections allow you to manipulate sections with greater precision.

If you do not specify a SECTIONS directive, the link step uses a default algorithm for combining and allocating the sections. Section 7.13, describes this algorithm in detail.

7.8.1 SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

SECTIONS

{

name : [*property* [**,** **property**] [**,** **property**] . . .]

name : [*property* [**,** **property**] [**,** **property**] . . .]

name : [*property* [**,** **property**] [**,** **property**] . . .]

}

Each section specification, beginning with *name*, defines an output section. (An output section is a section in the output file.) A section name can be a subsection specification. (See Section 7.8.4 for information on multi-level subsections.) After the section name is a list of properties that define the section's contents and how the section is allocated. The properties can be separated by optional commas. Possible properties for a section are as follows:

- **Load allocation** defines where in memory the section is to be loaded.

  Syntax: **load =** *allocation* or
  allocation or
  > *allocation*

- **Run allocation** defines where in memory the section is to be run.

  Syntax: **run =** *allocation* or
  run > *allocation*

- **Input sections** defines the input sections (object files) that constitute the output section.

  Syntax: **{** *input_sections* **}**

- **Section type** defines flags for special section types.

  Syntax: **type = COPY** or
  type = DSECT or
  type = NOLOAD

See Section 7.12.

- **Fill value** defines the value used to fill uninitialized holes.

  Syntax: **fill =** *value* or

  name : [*properties* **=** *value*]

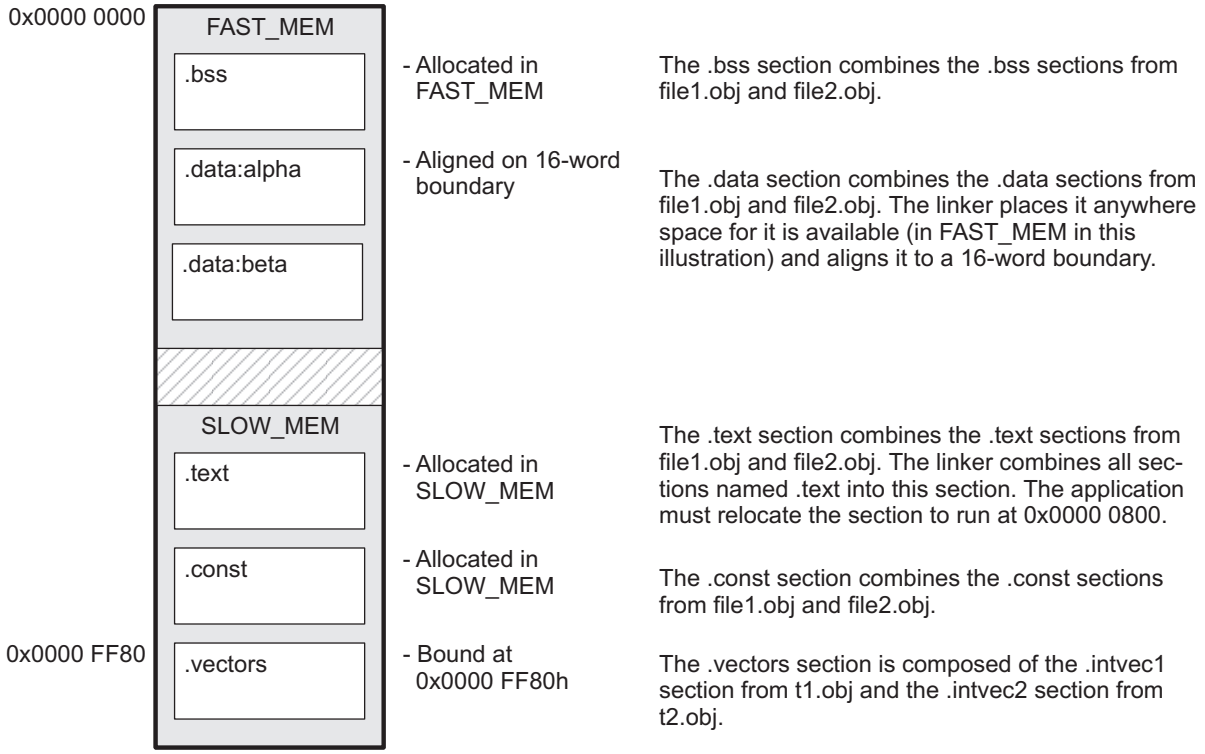
  See Section 7.15.

Example 7-4 shows a SECTIONS directive in a sample link command file.

Example 7-4. The SECTIONS Directive

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

SECTIONS
{
   .text:       load = SLOW_MEM, run = 0x00000800
   .const:      load = SLOW_MEM
   .bss:        load = FAST_MEM
   .vectors:    load = 0x0000FF80
      {
         t1.obj(.intvec1)
         t2.obj(.intvec2)
         endvec = .;
      }
   .data:alpha: align = 16
   .data:beta:  align = 16
}
```

Figure 7-3. Section Allocation Defined by Example 7-4



7.8.2 Allocation

The link step assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. The process of locating the output section in the target's memory and assigning its address(es) is called allocation. For more information about using separate load and run allocation, see Section 7.9.

If you do not tell the link step how a section is to be allocated, it uses a default algorithm to allocate the section. Generally, the link step puts sections wherever they fit into configured memory. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run allocation are separate, all parameters following the keyword LOAD apply to load allocation, and those following the keyword RUN apply to run allocation. The allocation parameters are:

Binding allocates a section at a specific address.

```
.text: load = 0x1000
```

Named memory allocates the section into a range defined in the MEMORY directive with the specified name (like SLOW_MEM) or attributes.

```
.text: load > SLOW_MEM
```

Alignment uses the align or palign keyword to specify that the section must start on an address boundary.

```
.text: align = 0x100
```

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

```
.text: block(0x100)
```

Page specifies the memory page to be used.

```
.text: load = SLOW_MEM PAGE 1
```

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

```
text: > SLOW_MEM            .text: {...} > SLOW_MEM
.text: > 0x4000
```

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

```
.text: > SLOW_MEM align 16 PAGE 2
```

Or if you prefer, use parentheses for readability:

```
.text: load = (SLOW_MEM align (16) page (2))
```

You can also use an input section specification to identify the sections from input files that are combined to form an output section. See Section 7.8.3.

7.8.2.1 Binding

You can supply a specific starting address for an output section by following the section name with an address:

```
.text: 0x00001000
```

This example specifies that the .text section must begin at location 0x1000. The binding address must be a 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 link step 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 link step issues an error message.

7.8.2.2 Named Memory

You can allocate a section into a memory range that is defined by the MEMORY directive (see Section 7.7). This example names ranges and links sections into them:

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

SECTIONS
{
   .text  :   > SLOW_MEM
   .data  :   > FAST_MEM ALIGN(128)
   .bss   :   > FAST_MEM
}
```

In this example, the link step places .text into the area called SLOW_MEM. The .data and .bss output sections are allocated into FAST_MEM. You can align a section within a named memory range; the .data section is aligned on a 128-byte boundary within the FAST_MEM range.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
   .text: > (X)   /* .text --> executable memory      */
   .data: > (RI)  /* .data --> read or init memory    */
   .bss : > (RW)  /* .bss --> read or write memory    */
}
```

In this example, the .text output section can be linked into either the SLOW_MEM or FAST_MEM area because both areas have the X attribute. The .data section can also go into either SLOW_MEM or FAST_MEM because both areas have the R and I attributes. The .bss output section, however, must go into the FAST_MEM area because only FAST_MEM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the link step uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no conflicting assignments exist, the .text section starts at address 0. If a section must start on a specific address, use binding instead of named memory.

7.8.2.3 Controlling Allocation Using The HIGH Location Specifier

The link step allocates output sections from low to high addresses within a designated memory range by default. Alternatively, you can cause the link step to allocate a section from high to low addresses within a memory range by using the HIGH location specifier in the SECTION directive declaration.

For example, given this MEMORY directive:

```
MEMORY
{
    RAM              : origin = 0x0200, length = 0x0800
    FLASH            : origin = 0x1100, length = 0xEEE0
    VECTORS          : origin = 0xFFE0, length = 0x001E
    RESET            : origin = 0xFFFE, length = 0x0002
}
```

and an accompanying SECTIONS directive:

```
SECTIONS
{
    .bss     : {} > RAM
    .sysmem  : {} > RAM
    .stack   : {} > RAM (HIGH)
}
```

The HIGH specifier used on the .stack section allocation causes the link step to attempt to allocate .stack into the higher addresses within the RAM memory range. The .bss and .sysmem sections are allocated into the lower addresses within RAM. Example 7-5 illustrates a portion of a map file that shows where the given sections are allocated within RAM for a typical program.

Example 7-5. Link Step Allocation With the HIGH Specifier

```
.bss       0    00000200    00000270     UNINITIALIZED
                  00000200    0000011a     rtsxxx.lib  : defs.obj (.bss)
                  0000031a    00000088                 : trgdrv.obj (.bss)
                  000003a2    00000078                 : lowlev.obj (.bss)
                  0000041a    00000046                 : exit.obj (.bss)
                  00000460    00000008                 : memory.obj (.bss)
                  00000468    00000004                 : _lock.obj (.bss)
                  0000046c    00000002                 : fopen.obj (.bss)
                  0000046e    00000002     hello.obj (.bss)

.sysmem    0    00000470    00000120     UNINITIALIZED
                  00000470    00000004     rtsxxx .lib : memory.obj (.sysmem)

.stack     0    000008c0    00000140     UNINITIALIZED
                  000008c0    00000002     rtsxxx .lib : boot.obj (.stack)
```

As shown in Example 7-5 , the .bss and .sysmem sections are allocated at the lower addresses of RAM (0x0200 - 0x0590) and the .stack section is allocated at address 0x08c0, even though lower addresses are available.

Without using the HIGH specifier, the link step allocation would have resulted in the code shown in Example 7-6

The HIGH specifier is ignored if it is used with specific address binding or automatic section splitting (>> operator).

Example 7-6. Link Step Allocation Without HIGH Specifier

```
.bss       0    00000200    00000270     UNINITIALIZED
                  00000200    0000011a     rtsxxx.lib  : defs.obj (.bss)
                  0000031a    00000088                 : trgdrv.obj (.bss)
                  000003a2    00000078                 : lowlev.obj (.bss)
                  0000041a    00000046                 : exit.obj (.bss)
                  00000460    00000008                 : memory.obj (.bss)
                  00000468    00000004                 : _lock.obj (.bss)
                  0000046c    00000002                 : fopen.obj (.bss)
                  0000046e    00000002     hello.obj (.bss)

.stack     0    00000470    00000140     UNINITIALIZED
                  00000470    00000002     rtsxxx.lib  : boot.obj (.stack)

.sysmem    0    000005b0    00000120     UNINITIALIZED
                  000005b0    00000004     rtsxxx.lib  : memory.obj (.sysmem)
```

7.8.2.4 Alignment and Blocking

You can tell the link step to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the align keyword. For example, the following code allocates .text so that it falls on a 32-byte boundary:

```
.text: load = align(32)
```

You can specify the same alignment with the palign keyword. In addition, palign ensures the section's size is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

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

```
bss: load = block(0x0080)
```

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

7.8.2.5 Using the Page Method

Using the page method of specifying an address, you can allocate a section into an address space that is named in the MEMORY directive. For example:

```
MEMORY
{
   PAGE 0 :  PROG    : origin = 0x00000800,   length = 0x00240
   PAGE 1 :  DATA    : origin = 0x00000A00,   length = 0x02200
   PAGE 1 :  OVR_MEM : origin = 0x00002D00,   length = 0x01000
   PAGE 2 :  DATA    : origin = 0x00000A00,   length = 0x02200
   PAGE 2 :  OVR_MEM : origin = 0x00002D00,   length = 0x01000
}
SECTIONS
{
   .text:   PAGE = 0
   .data:   PAGE = 2
   .cinit:  PAGE = 0
```

```
   .bss:    PAGE = 1
}
```

In this example, the .text and .cinit sections are allocated to PAGE 0. They are placed anywhere within the bounds of PAGE 0. The .data section is allocated anywhere within the bounds of PAGE 2. The .bss section is allocated anywhere within the bounds of PAGE 1.

You can use the page method in conjunction with any of the other methods to restrict an allocation to a specific address space. For example:

```
.text: load = OVR_MEM PAGE 1
```

In this example, the .text section is allocated to the named memory range OVR_MEM. There are two named memory ranges called OVR_MEM, however, so you must specify which one is to be used. By adding PAGE 1, you specify the use of the OVR_MEM memory range in address space PAGE 1 rather than in address space PAGE 2.

7.8.3 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. In general, the link step combines input sections by concatenating them in the order in which they are specified. However, if alignment or blocking is specified for an input section, all of the input sections within the output section are ordered as follows:

- All aligned sections, from largest to smallest
- All blocked sections, from largest to smallest
- All other sections, from largest to smallest

The size of an output section is the sum of the sizes of the input sections that it comprises.

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

Example 7-7. The Most Common Method of Specifying Section Contents

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

In Example 7-7, the link step takes all the .text sections from the input files and combines them into the .text output section. The link step concatenates the .text input sections in the order that it encounters them in the input files. The link step performs similar operations with the .data and .bss sections. You can use this type of specification for any output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
  .text :              /* Build .text output section        */
  {
    f1.obj(.text)      /* Link .text section from f1.obj    */
    f2.obj(sec1)       /* Link sec1 section from f2.obj     */
    f3.obj             /* Link ALL sections from f3.obj     */
    f4.obj(.text,sec2) /* Link .text and sec2 from f4.obj   */
   }
}
```

It is not necessary for input sections to have the same name as each other or as the output section they become part of. If a file is listed with no sections,*all* of its sections are included in the output section. If any additional input sections have the same name as an output section but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the link step found more .text sections in the preceding example and these .text sections *were not* specified anywhere in the SECTIONS directive, the link step would concatenate these extra sections after f4.obj(sec2).

The specifications in Example 7-7 are actually a shorthand method for the following:

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

The specification *(.text) means *the unallocated .text sections from all the input files.* This format is useful when:

- You want the output section to contain all input sections that have a specified name, but the output section name is different from the input sections' name.
- You want the link step to allocate the input sections before it processes additional input sections or commands within the braces.

The following example illustrates the two purposes above:

```
SECTIONS
{
   .text   :  {
                     abc.obj(xqt)
                        *(.text)
              }
   .data   :  {
                     *(.data)
                     fil.obj(table)
                }
}
```

In this example, the .text output section contains a named section xqt from file abc.obj, which is followed by all the .text input sections. The .data section contains all the .data input sections, followed by a named section table from the file fil.obj. This method includes all the unallocated sections. For example, if one of the .text input sections was already included in another output section when the link step encountered *(.text), the link step could not include that first .text input section in the second output section.

7.8.4 Using Multi-Level Subsections

Originally, subsections were identified with the base section name and a subsection name separated by a colon. For example, A:B names a subsection of the base section A. In certain places in a link command file specifying a base name, such as A, selects the section A as well as any subsections of A, such as A:B or A:C.

This concept has been extended to include multiple levels of subsection naming. The original constraints are still true, but a name such as A:B can be used to specify a (sub)section of that name as well as any (multi-level) subsections beginning with that name, such as A:B:C, A:B:OTHER, etc. All the subsections of A:B are also subsections of A. A and A:B are supersections of A:B:C. Among a group of supersections of a subsection, the nearest supersection is the supersection with the longest name. Thus, among {A, A:B} the nearest supersection of A:B:C:D is A:B.

With multiple levels of subsections, the constraints are the following:

1. When specifying **input** sections within a file (or library unit) the section name selects an input section of the same name and any subsections of that name.
2. Input sections that are not explicitly allocated are allocated in an existing **output** section of the same name or in the nearest existing supersection of such an output section. An exception to this rule is that during a partial link (specified by the --relocatable link step 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 link step-generated output section "europe" */
}
```

Upward Compatibility of Multi-Level Subsections

Note: Existing link step commands that use the existing single-level subsection features and which do not contain section names containing multiple colon characters continue to behave as before. However, if section names in a link command file or in the input sections supplied to the link step contain multiple colon characters, some change in behavior could be possible. You should carefully consider the impact of the new rules for multiple levels to see if it affects a particular system link.

7.8.5 Allocation Using Multiple Memory Ranges

The link step allows you to specify an explicit list of memory ranges into which an output section can be allocated. Consider the following example:

```
MEMORY
{
   P_MEM1 :  origin = 02000h,  length = 01000h
   P_MEM2 :  origin = 04000h,  length = 01000h
   P_MEM3 :  origin = 06000h,  length = 01000h
   P_MEM4 :  origin = 08000h,  length = 01000h
}

SECTIONS
{
   .text  : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

The | operator is used to specify the multiple memory ranges. The .text output section is allocated as a whole into the first memory range in which it fits. The memory ranges are accessed in the order specified. In this example, the link step first tries to allocate the section in P_MEM1. If that attempt fails, the link step 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 link step issues an error message.

With this type of SECTIONS directive specification, the link step can seamlessly handle an output section that grows beyond the available space of the memory range in which it is originally allocated. Instead of modifying the link command file, you can let the link step move the section into one of the other areas.

7.8.6 Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges

The link step can split output sections among multiple memory ranges to achieve an efficient allocation. Use the >> operator to indicate that an output section can be split, if necessary, into the specified memory ranges. For example:

```
MEMORY
{
   P_MEM1 :  origin = 02000h,  length = 01000h
   P_MEM2 :  origin = 04000h,  length = 01000h
   P_MEM3 :  origin = 06000h,  length = 01000h
   P_MEM4 :  origin = 08000h,  length = 01000h
}

SECTIONS
{
  .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

In this example, the >> operator indicates that the .text output section can be split among any of the listed memory areas. If the .text section grows beyond the available memory in P_MEM1, it is split on an input section boundary, and the remainder of the output section is allocated to P_MEM2 | P_MEM3 | P_MEM4.

The | operator is used to specify the list of multiple memory ranges.

You can also use the >> operator to indicate that an output section can be split within a single memory range. This functionality is useful when several output sections must be allocated into the same memory range, but the restrictions of one output section cause the memory range to be partitioned. Consider the following example:

```
MEMORY
{
   RAM :  origin = 01000h,  length = 08000h
}

SECTIONS
{
  .special: { f1.obj(.text) } = 04000h
  .text: { *(.text) } >> RAM
}
```

The .special output section is allocated near the middle of the RAM memory range. This leaves two unused areas in RAM: from 01000h to 04000h, and from the end of f1.obj(.text) to 08000h. The specification for the .text section allows the link step to split the .text section around the .special section and use the available space in RAM on either side of .special.

The >> operator can also be used to split an output section among all memory ranges that match a specified attribute combination. For example:

```
MEMORY
{
  P_MEM1 (RWX) : origin = 01000h,  length = 02000h
  P_MEM2 (RWI) : origin = 04000h,  length = 01000h
}

SECTIONS
{
  .text: { *(.text) } >> (RW)
}
```

The link step attempts to allocate all or part of the output section into any memory range whose attributes match the attributes specified in the SECTIONS directive.

This SECTIONS directive has the same effect as:

```
SECTIONS
{
.text: { *(.text) } >> P_MEM1 | P_MEM2}
```

```
}
```

Certain sections should not be split:

- Certain sections created by the compiler, including
  - The .cinit section, which contains the autoinitialization table for C/C++ programs
  - The .pinit section, which contains the list of global constructors for C++ programs
  - The .bss section, which defines global variables
- An output section with an input section specification that includes an expression to be evaluated. The expression may define a symbol that is used in the program to manage the output section at run time.
- An output section that 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 link step issues a warning and ignores the operator.

7.8.7 Allocating an Archive Member to an Output Section

The ability to specify an archive member of a library archive for allocation into a specific output section can be specified inside angle brackets after a library name. Any object files separated by commas or spaces from the specified archive file are legal within the angle brackets. The syntax for allocating archived library members specifically inside of a SECTIONS directive is as follows:

[**--library**=] *library name* **<***member1*[,] *member2*[,] *...***>** [**(***input sections***)**]

Example 7-8 specifies that the text sections of boot.obj, exit.obj, and strcpy.obj from the run-time-support library should be placed in section .boot. The remainder of the .text sections from the run-time-support library are to be placed in section .rts. Finally, the remainder of all other .text sections are to be placed in section .text.

Example 7-8. Archive Members to Output Sections

```
SECTIONS
{
        boot    >        BOOT1
        {
                --library=rtsXX.lib<boot.obj> (.text)
                --library=rtsXX.lib>exit.obj strcpy.obj> (.text)
        }

        .rts    >        BOOT2
        {
                --library=rtsXX.lib (.text)
        }

        .text    >        RAM
        {
                * (.text)
        }
}
```

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 7-8 is optional when listing specific archive members inside < >. Using < > implies that you are referring to a library.

To collect a set of the input sections from a library in one place, use the --library option within the SECTIONS directive. For example, the following collects all the .text sections from rts2800.lib into the .rtstest section:

```
SECTIONS
{
```

```
   .rtstest { ---library=rts2800.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 link step searches to resolve references. If you use the --priority option, the first library specified in the command file will be searched first.

7.9 Specifying a Section's Run-Time Address

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

The link step provides a simple way to accomplish this. You can use the SECTIONS directive to direct the link step to allocate a section twice: once to set its load address and again to set its run address. For example:

```
.fir: load = SLOW_MEM, run = FAST_MEM
```

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

See Section 2.5 for an overview on run-time relocation.

7.9.1 Specifying Load and Run Addresses

The load address determines where a loader places the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this does *not* happen automatically when you specify a separate run address.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is allocated as if it were two sections of the same size. This means that both allocations occupy space in the memory map and cannot overlay each other or other sections. (The UNION directive provides a way to overlay sections; see Section 7.10.1.)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. Everything related to allocation after the keyword *load* affects the load address until the keyword *run* is seen, after which, everything affects the run address. The load and run allocations are completely independent, so any qualification of one (such as alignment) has no effect on the other. You can also specify run first, then load. Use parentheses to improve readability.

The examples below specify load and run addresses:

```
.data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

(align applies only to load)

```
.data: load = (SLOW_MEM align 32), run = FAST_MEM
```

(identical to previous example)

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

(align 32 in FAST_MEM for run; align 16 anywhere for load)

7.9.2 Uninitialized Sections

Uninitialized sections (such as .bss) are not loaded, so their only significant address is the run address. The link step allocates uninitialized sections only once: if you specify both run and load addresses, the link step warns you and ignores the load address. Otherwise, if you specify only one address, the link step treats it as a run address, regardless of whether you call it load or run. This example specifies load and run addresses for an uninitialized section:

```
.bss: load = 0x1000, run = FAST_MEM
```

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

```
.bss: load = FAST_MEM
.bss: run = FAST_MEM
.bss: > FAST_MEM
```

7.9.3 Referring to the Load Address by Using the .label Directive

Normally, any reference to a symbol in a section refers to its run-time address. However, it may be necessary at run time to refer to a load-time address. Specifically, the code that copies a section from its load address to its run address must have access to the load address. The .label directive defines a special symbol that refers to the section's load address. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address. See Create a Load-Time Address Label for more information on the .label directive.

Example 7-9 and Example 7-10 show the use of the .label directive to copy a section from its load address in SLOW_MEM to its run address in FAST_MEM. Figure 7-4 illustrates the run-time execution of Example 7-9.

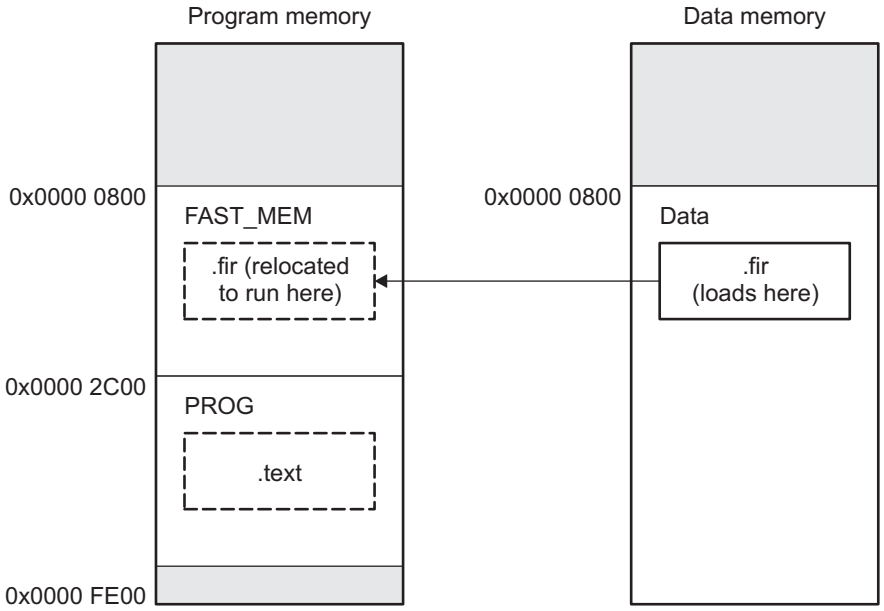
Example 7-9. Copying Section Assembly Language File

```
;----------------------------------------------------------
;  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 RAM
;----------------------------------------------------------
     B  fir
```

Example 7-10. Link Step Command File for Example 7-9

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

MEMORY
{
   PAGE 0 :  RAM       :  origin = 0x00000800,  length = 0x00002400
   PAGE 0 :  SLOW_MEM  :  origin = 0x00002C00,  length = 0x0000D200
   PAGE 1 :  FAST_MEM  :  origin = 0x00000800,  length = 0x0000F800
}

SECTIONS
{
   .text: load = SLOW_MEM PAGE 0
   .fir:  load = FAST_MEM PAGE 1, run RAM PAGE 0
}
```

Figure 7-4. Run-Time Execution of Example 7-9



7.10 Using UNION and GROUP Statements

Two SECTIONS statements allow you to conserve memory: GROUP and UNION. Unioning sections causes the link step to allocate them to the same run address. Grouping sections causes the link step to allocate them contiguously in memory. Section names can refer to sections, subsections, or archive library members.

7.10.1 Overlaying Sections With the UNION Statement

For some applications, you may want to allocate more than one section to run at the same address. For example, you may have several routines you want in fast external memory at various stages of execution. Or you may want several data objects that are not active at the same time to share a block of memory. The UNION statement within the SECTIONS directive provides a way to allocate several sections at the same run-time address.

In Example 7-11, the .bss sections from file1.obj and file2.obj are allocated at the same address in FAST_MEM. In the memory map, the union occupies as much space as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

Example 7-11. The UNION Statement

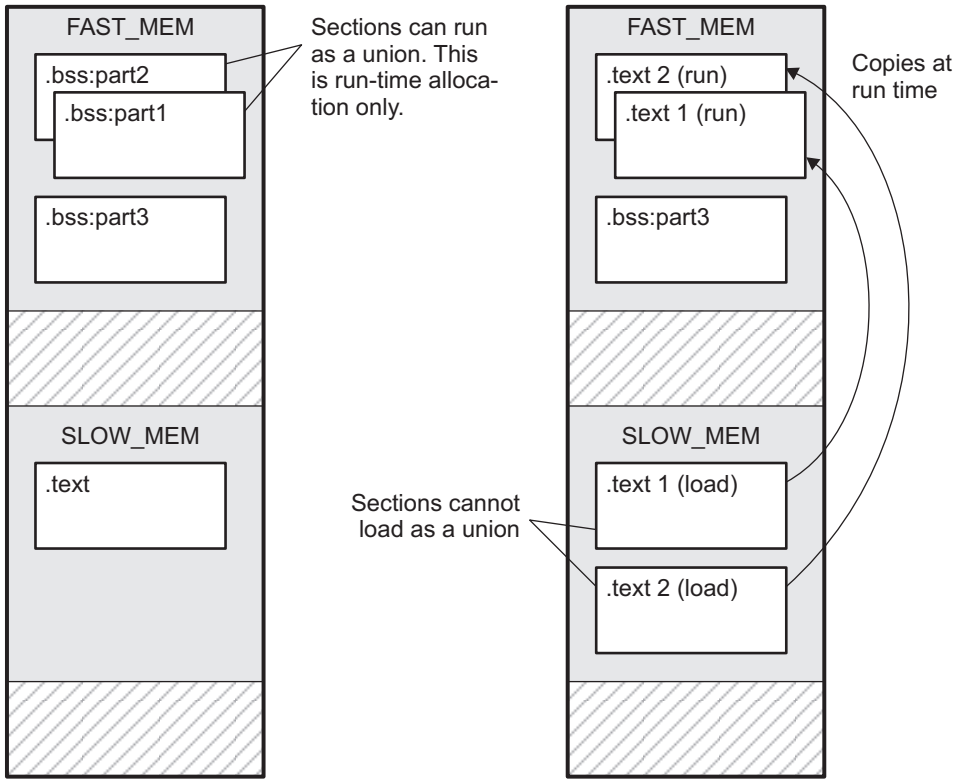
```
SECTIONS
{
   .text: load = SLOW_MEM
   UNION: run  = FAST_MEM
   {
      .bss:part1: { file1.obj(.bss) }
      .bss:part2: { file2.obj(.bss) }
   }
      .bss:part3: run = FAST_MEM { globals.obj(.bss) }
}
```

Allocation of a section as part of a union affects only its *run address.* Under no circumstances can sections be overlaid for loading. If an initialized section is a union member (an initialized section, such as .text, has raw data), its load allocation *must* be separately specified. See Example 7-12.

Example 7-12. Separate Load Addresses for UNION Sections

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

Figure 7-5. Memory Allocation Shown in Example 7-11 and Example 7-12



Since the .text sections contain data, they cannot *load* as a union, although they can be *run* as a union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a UNION, the link step 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 link step issues a warning and ignores the load address.

7.10.2 Grouping Output Sections Together

The SECTIONS directive's GROUP option forces several output sections to be allocated contiguously. For example, assume that a section named term_rec contains a termination record for a table in the .data section. You can force the link step to allocate .data and term_rec together:

Example 7-13. Allocate Sections Together

```
SECTIONS
{
   .text           /* Normal output section             */
   .bss            /* Normal output section             */
   GROUP  0x00001000 : /* Specify a group of sections    */
   {
      .data        /* First section in the group        */
      term_rec     /* Allocated immediately after .data */
   }
}
```

You can use binding, alignment, or named memory to allocate a GROUP in the same manner as a single output section. In the preceding example, the GROUP is bound to address 0x00001000. This means that .data is allocated at 0x00001000, and term_rec follows it in memory.

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.

7.10.3 Nesting UNIONs and GROUPs

The link step allows arbitrary nesting of GROUP and UNION statements with the SECTIONS directive. By nesting GROUP and UNION statements, you can express hierarchical overlays and groupings of sections. Example 7-14 shows how two overlays can be grouped together.

Example 7-14. Nesting GROUP and UNION Statements

```
SECTIONS
{
   GROUP 1000h : run = FAST_MEM
   {
      UNION:
      {
         mysect1: load = SLOW_MEM
         mysect2: load = SLOW_MEM
      }
      UNION:
      {
         mysect3: load = SLOW_MEM
         mysect4: load = SLOW_MEM
      }
   }
}
```

For this example, the link step performs the following allocations:

- The four sections (mysect1, mysect2, mysect3, mysect4) are assigned unique, non-overlapping load addresses in the SLOW_MEM memory region. This assignment is determined by the particular load allocations given for each section.
- Sections mysect1 and mysect2 are assigned the same run address in FAST_MEM.
- Sections mysect3 and mysect4 are assigned the same run address in FAST_MEM.
- The run addresses of mysect1/mysect2 and mysect3/mysect4 are allocated contiguously, as directed by the GROUP statement (subject to alignment and blocking restrictions).

To refer to groups and unions, link step diagnostic messages use the notation:

GROUP_*n* UNION_*n*

In this notation, *n* is a sequential number (beginning at 1) that represents the lexical ordering of the group or union in the link step control file, without regard to nesting. Groups and unions each have their own counter.

7.10.4 Checking the Consistency of Allocators

The link step 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 link step uses the run address of the top-level structure to compute the run addresses of the components within groups and unions.
- The link step does not accept a load allocation for UNIONs.

- The link step does not accept a load allocation for uninitialized sections.
- In most cases, you must provide a load allocation for an initialized section. However, the link step 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 link step issues a diagnostic message to request that these load allocations be specified explicitly.

7.11 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 link step to load various output sections into each of these banks or into banks that are not mapped at load time.

The link step 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 7.10.1) sound similar because they both deal with overlays. They are, in fact, quite different. UNION allows multiple sections to be overlaid within the same memory space. Overlay pages, on the other hand, define multiple memory spaces. It is possible to use the page facility to approximate the function of UNION, but it is cumbersome.

7.11.1 Using the MEMORY Directive to Define Overlay Pages

To the link step, 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 link step allocates initialized sections into PAGE 0 (program memory) and uninitialized sections into PAGE 1 (data memory).

7.11.2 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 7-15 shows how you use the MEMORY directive to obtain this configuration:

Example 7-15. MEMORY Directive With Overlay Pages

```
MEMORY
{
   PAGE 0  : RAM      :origin = 0x00000800, length = 0x00000240
           : PROG     :origin = 0x00002C00, length = 0x0000D200
   PAGE 1  : OVR_MEM  :origin = 0x00000A00, length = 0x00002200
           : DATA     :origin = 0x00002C00, length = 0x0000D400
   PAGE 2  : OVR_MEM  :origin = 0x00000A00, length = 0x00002200
}
```

Example 7-15 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.

7.11.3 Using Overlay Pages With the SECTIONS Directive

Assume that you are using the MEMORY directive as shown in Example 7-15. 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 7-16 shows how to use the SECTIONS directive overlays to accomplish these objectives.

Example 7-16. SECTIONS Directive Definition for Overlays in Example 7-10

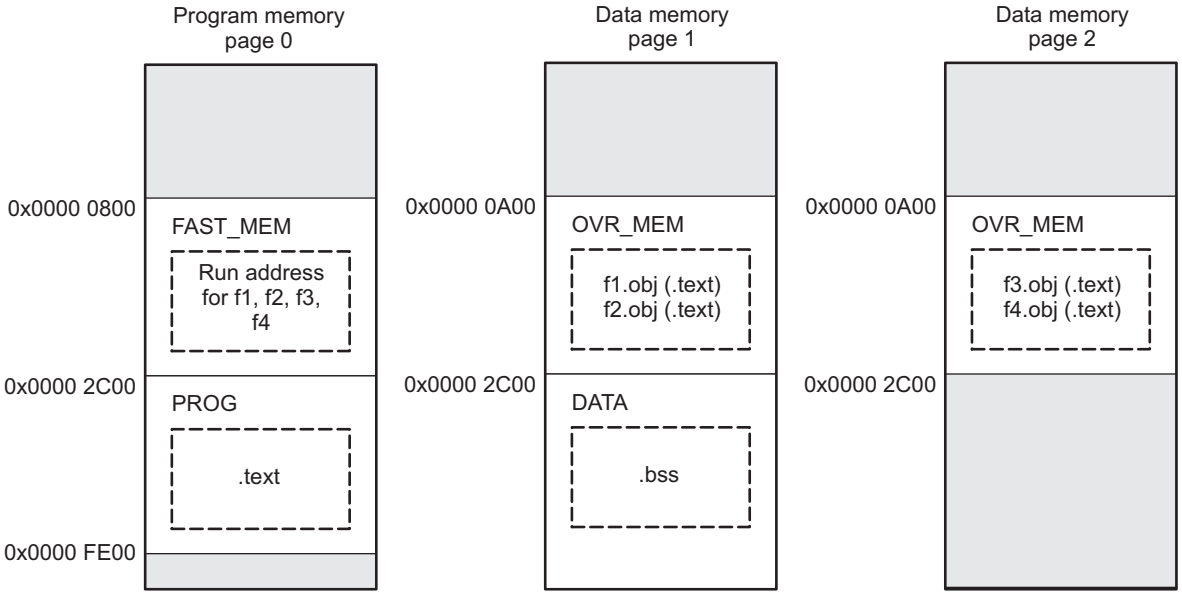
```
SECTIONS
{
   UNION :  run = RAM
   {
      S1 :  load = OVR_MEM PAGE 1
      {
         s1_load = 0x00000A00h;
         s1_start = .;
         f1.obj (.text)
         f2.obj (.text)
         s1_length = . - s1_start;
      }
      S2 :  load = OVR_MEM PAGE 2
      {
         s2_load = 0x00000A00h;
         s2_start = .;
         f3.obj (.text)
         f4.obj (.text)
         s2_length = . - s2_start;
      }
   }

   .text: load = PROG PAGE 0
   .data: load = PROG PAGE 0
   .bss : load = DATA PAGE 1
}
```

The four modules are f1, f2, f3, and f4. Modules f1 and f2 are combined into output section S1, and f3 and f4 are combined into output section S2. The PAGE specifications for S1 and S2 tell the link step 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.

7.11.4 Memory Allocation for Overlaid Pages

Figure 7-6 shows overlay pages defined by the MEMORY directive in Example 7-15 and the SECTIONS directive in Example 7-16.

Figure 7-6. Overlay Pages Defined in Example 7-15 and Example 7-16



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

You can assign three special types to output sections: DSECT, COPY, and NOLOAD. These types affect the way that the program is treated when it is linked and loaded. You can assign a type to a section by placing the type after the section definition. For example:

```
SECTIONS
{
   sec1: load = 0x00002000, type =  DSECT  {f1.obj}
   sec2: load = 0x00004000, type =  COPY   {f2.obj}
   sec3: load = 0x00006000, type =  NOLOAD {f3.obj}
}
```

- The DSECT type creates a dummy section with the following characteristics:
  - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.
  - It can overlay other output sections, other DSECTs, and unconfigured memory.
  - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.
  - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.
  - The section's contents, relocation information, and line number information are not placed in the output module.

  In the preceding example, none of the sections from f1.obj are allocated, but all the symbols are relocated as though the sections were linked at address 0x2000. The other sections can refer to any of the global symbols in sec1.

- A COPY section is similar to a DSECT section, except that its contents and associated information are written to the output module. The .cinit section that contains initialization tables for the 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 link step allocates space for the section, and it appears in the memory map listing.

7.13 Default Allocation Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections that you choose *not* to specify must still be handled by the link step. The link step uses default algorithms to build and allocate sections within the specifications you supply.

If you do not use the MEMORY and SECTIONS directives, the link step allocates output sections as though the definitions in Example 7-17 were specified.

Example 7-17. 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 */
   .bss:     PAGE = 1
}
```

All .text input sections are concatenated to form a .text output section in the executable output file, and all .data input sections are combined to form a .data output section.

If you use a SECTIONS directive, the link step performs *no part* of the default allocation. Allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described next in Section 7.13.1.

7.13.1 How the Allocation Algorithm Creates Output Sections

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

Method 1 As the result of a SECTIONS directive definition

Method 2 By combining input sections with the same name into an output section that is not defined in a SECTIONS directive

If an output section is formed as a result of a SECTIONS directive, this definition completely determines the section's contents. (See Section 7.8 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 link step combines all such input sections that have the same name into an output section with that name. For example, suppose the files f1.obj and f2.obj both contain named sections called Vectors and that the SECTIONS directive does not define an output section for them. The link step 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 link step 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 link step option (see Section 7.4.20) to cause the link step to display a message when it creates a new output section.

After the link step 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 link step uses the default configuration as shown in Example 7-17. (See Section 7.7 for more information on configuring memory.)

7.13.2 Reducing Memory Fragmentation

The link step's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. The algorithm comprises these steps:

1. Each output section for which you have supplied a specific binding address is placed in memory at that address.
2. Each output section that is included in a specific, named memory range or that has memory attribute restrictions is allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.
3. Any remaining sections are allocated in the order in which they are defined. Sections not defined in a SECTIONS directive are allocated in the order in which they are encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

7.14 Assigning Symbols at Link Time

Link step assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to initialize a variable or pointer to an allocation-dependent value.

7.14.1 Syntax of Assignment Statements

The syntax of assignment statements in the link step 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 link step defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in Section 7.14.3. Assignment statements *must* terminate with a semicolon.

The link step 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 link step assignment statement to assign cur_tab at link time:

```
prog.obj           /* Input file                          */
cur_tab = Table1; /* Assign cur_tab to one of the tables */
```

7.14.2 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 link step's . symbol is analogous to the assembler's $ symbol. The . symbol can be used only in assignment statements within a SECTIONS directive because . is meaningful only during allocation and SECTIONS controls the allocation process. (See Section 7.8.)

The . symbol refers to the current run address, not the current load address, of the section.

For example, suppose a program needs to know the address of the beginning of the .data section. By using the .global directive (see Identify Global Symbols), you can create an external undefined variable called Dstart in the program. Then, assign the value of . to Dstart:

```
SECTIONS
{
   .text:    {}
   .data:    {Dstart = .;}
   .bss :    {}
}
```

This defines Dstart to be the first linked address of the .data section. (Dstart is assigned *before* .data is allocated.) The link step relocates all references to Dstart.

A special type of assignment assigns a value to the . symbol. This adjusts the SPC within an output section and creates a hole between two input sections. Any value assigned to . to create a hole is relative to the beginning of the section, not to the address actually represented by the . symbol. Holes and assignments to . are described in Section 7.15.

7.14.3 Assignment Expressions

These rules apply to link step expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in Table 7-2.
- All numbers are treated as long (32-bit) integers.
- Constants are identified by the link step 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.
- Link step 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 link step supports the C language operators listed in Table 7-2 in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in Table 7-2, the link step 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 7-2. Groups of Operators Used in Expressions (Precedence)

Group 1 (Highest Precedence)		Group 6	
!	Logical NOT	&	Bitwise AND
~	Bitwise NOT		
-	Negation		
Group 2		**Group 7**	

Table 7-2. Groups of Operators Used in Expressions (Precedence) (continued)

*	Multiplication	|	Bitwise OR
/	Division		
%	Modulus		
Group 3		**Group 8**	
+	Addition	&&	Logical AND
-	Subtraction		
Group 4		**Group 9**	
>>	Arithmetic right shift	||	Logical OR
<<	Arithmetic left shift		
Group 5		**Group 10 (Lowest Precedence)**	
==	Equal to	=	Assignment
! =	Not equal to	+ =	A + = B " A = A + B
>	Greater than	- =	A - = B " A = A - B
<	Less than	* =	A * = B " A = A * B
< =	Less than or equal to	/ =	A / = B " A = A / B
> =	Greater than or equal to		

7.14.4 Symbols Defined by the Link Step

The link step automatically defines several symbols based on which sections are used in your assembly source. A program can use these symbols at run time to determine where a section is linked. Since these symbols are external, they appear in the link step map. Each symbol can be accessed in any assembly language module if it is declared with a .global directive (see Identify Global Symbols). You must have used the corresponding section in a source module for the symbol to be created. Values are assigned to these symbols as follows:

.text is assigned the first address of the .text output section. (It marks the *beginning* of executable code.)

etext is assigned the first address following the .text output section. (It marks the *end* of executable code.)

.data is assigned the first address of the .data output section. (It marks the *beginning* of initialized data tables.)

edata is assigned the first address following the .data output section. (It marks the *end* of initialized data tables.)

.bss is assigned the first address of the .bss output section. (It marks the *beginning* of uninitialized data.)

end is assigned the first address following the .bss output section. (It marks the *end* of uninitialized data.)

The following symbols are defined only for C/C++ support when the --ram_model or --rom_model option is used.

__STACK_SIZE is assigned the size of the .stack section.

__SYSMEM_SIZE is assigned the size of the .sysmem section.

7.14.5 Assigning Exact Start, End, and Size Values of a Section to a Symbol

The code generation tools currently support the ability to load program code in one area of (slow) memory and run it in another (faster) area. This is done by specifying separate load and run addresses for an output section or group in the link command file. Then execute a sequence of instructions (the copying code in Example 7-9) that moves the program code from its load area to its run area before it is needed.

There are several responsibilities that a programmer must take on when setting up a system with this feature. One of these responsibilities is to determine the size and run-time address of the program code to be moved. The current mechanisms to do this involve use of the .label directives in the copying code. A simple example is illustrated Example 7-9.

This method of specifying the size and load address of the program code has limitations. While it works fine for an individual input section that is contained entirely within one source file, this method becomes more complicated if the program code is spread over several source files or if the programmer wants to copy an entire output section from load space to run space.

Another problem with this method is that it does not account for the possibility that the section being moved may have an associated far call trampoline section that needs to be moved with it.

7.14.6 Why the Dot Operator Does Not Always Work

The dot operator (.) is used to define symbols at link-time with a particular address inside of an output section. It is interpreted like a PC. Whatever the current offset within the current section is, that is the value associated with the dot. Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.obj(.text)
        end_of_s2 = .;
}
```

This statement creates three symbols:

- end_of_s1—the end address of .text in s1.obj
- start_of_s2—the start address of .text in s2.obj
- end_of_s2—the end address of .text in s2.obj

Suppose there is padding between s1.obj and s2.obj that is created as a result of alignment. Then start_of_s2 is not really the start address of the .text section in s2.obj, but it is the address before the padding needed to align the .text section in s2.obj. This is due to the link step's interpretation of the dot operator as the current PC. It is also due to the fact that the dot operator is evaluated independently of the input sections around it.

Another potential problem in the above example is that end_of_s2 may not account for any padding that was required at the end of the output section. You cannot reliably use end_of_s2 as the end address of the output section. One way to get around this problem is to create a dummy section immediately after the output section in question. For example:

```
GROUP
{
   outsect:
   {
      start_of_outsect = .;
      ...
   }
      dummy: { size_of_outsect = . - start_of_outsect; }
}
```

7.14.7 Address and Dimension Operators

Six new operators have been added to the link command file syntax:

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

Link Step Command File Operator Equivalencies

Note: LOAD_START() and START() are equivalent, as are LOAD_END()/END() and LOAD_SIZE()/SIZE().

The new address and dimension operators can be associated with several different kinds of allocation units, including input items, output sections, GROUPs, and UNIONs. The following sections provide some examples of how the operators can be used in each case.

7.14.7.1 Input Items

Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.obj(.text)
        end_of_s2 = .;
}
```

This can be rewritten using the START and END operators as follows:

```
outsect:
{
        s1.obj(.text) { END(end_of_s1) }
        s2.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

The values of end_of_s1 and end_of_s2 will be the same as if you had used the dot operator in the original example, but start_of_s2 would be defined after any necessary padding that needs to be added between the two .text sections. Remember that the dot operator would cause start_of_s2 to be defined before any necessary padding is inserted between the two input sections.

The syntax for using these operators in association with input sections calls for braces { } to enclose the operator list. The operators in the list are applied to the input item that occurs immediately before the list.

7.14.7.2 Output Section

The START, END, and SIZE operators can also be associated with an output section. Here is an example:

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
        <list of input items>
}
```

In this case, the SIZE operator defines size_of_outsect to incorporate any padding that is required in the output section to conform to any alignment requirements that are imposed.

The syntax for specifying the operators with an output section do not require braces to enclose the operator list. The operator list is simply included as part of the allocation specification for an output section.

7.14.7.3 GROUPs

Here is another use of the START and SIZE operators in the context of a GROUP specification:

```
GROUP
{
        outsect1: { ... }
        outsect2: { ... }
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

This can be useful if the whole GROUP is to be loaded in one location and run in another. The copying code can use group_start and group_size as parameters for where to copy from and how much is to be copied. This makes the use of .label in the source code unnecessary.

7.14.7.4 UNIONs

The RUN_SIZE and LOAD_SIZE operators provide a mechanism to distinguish between the size of a UNION's load space and the size of the space where its constituents are going to be copied before they are run. Here is an example:

```
UNION: run = RAM, LOAD_START(union_load_addr),
       LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
        .text1: load = ROM, SIZE(text1_size) { f1.obj(.text) }
        .text2: load = ROM, SIZE(text2_size) { f2.obj(.text) }
}
```

Here union_ld_sz is going to be equal to the sum of the sizes of all output sections placed in the union. The union_run_sz value is equivalent to the largest output section in the union. Both of these symbols incorporate any padding due to blocking or alignment requirements.

7.15 Creating and Filling Holes

The link step 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 link step handles holes and how you can fill holes (and uninitialized sections) with values.

7.15.1 Initialized and Uninitialized Sections

There are two rules to remember about the contents of output sections. An output section contains either:

- Raw data for the *entire* section
- *No* raw data

A section that has raw data is referred to as *initialized*. This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections *always* have raw data if anything was assembled into them. Named sections defined with the .sect assembler directive also have raw data.

By default, the .bss section (see Reserve Space in the .bss Section) and sections defined with the .usect directive (see Reserve Uninitialized Space) have no raw data (they are *uninitialized*). They occupy space in the memory map but have no actual contents. Uninitialized sections typically reserve space in fast external memory for variables. In the object file, an uninitialized section has a normal section header and can have symbols defined in it; no memory image, however, is stored in the section.

7.15.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the link step to leave extra space between input sections within an output section. When such a hole is created, *the link step must supply raw data for the hole.*

Holes can be created only *within* output sections. Space can exist *between* output sections, but such space is not a hole. To fill the space between output sections, see Section 7.7.2.

To create a hole in an output section, you must use a special type of link step assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by .) by adding to it, assigning a greater value to it, or aligning it on an address boundary. The operators, expressions, and syntaxes of assignment statements are described in Section 7.14.

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

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
     . += 0x0100    /* Create a hole with size 0x0100  */
      file2.obj(.text)
    . = align(16);  /* Create a hole to align the SPC */
      file3.obj(.text)
    }
}
```

The output section outsect is built as follows:

1. The .text section from file1.obj is linked in.
2. The link step creates a 256-byte hole.
3. The .text section from file2.obj is linked in after the hole.
4. The link step creates another hole by aligning the SPC on a 16-byte boundary.
5. Finally, the .text section from file3.obj is linked in.

All values assigned to the . symbol within a section refer to the *relative address within the section.* The link step handles assignments to the . symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement . = align(16) in the example. This statement effectively aligns the file3.obj .text section to start on a 16-byte boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, the file3.obj .text section will not be aligned either.

The . symbol refers to the current run address, not the current load address, of the section.

Expressions that decrement the . symbol are illegal. For example, it is invalid to use the -= operator in an assignment to the . symbol. The most common operators used in assignments to the . symbol are += and align.

If an output section contains all input sections of a certain type (such as .text), you can use the following statements to create a hole at the beginning or end of the output section.

```
    .text:   {  .+= 0x0100; }     /* Hole at the beginning */
    .data:   {  *(.data)
                . += 0x0100; }    /* Hole at the end       */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the link step treats the uninitialized section as a hole and supplies data for it.* The following example illustrates this method:

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
file1.obj(.bss)           /* This becomes a hole */
   }
}
```

Because the .text section has raw data, all of outsect must also contain raw data. Therefore, the uninitialized .bss section becomes a hole.

Uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

7.15.3 Filling Holes

When a hole exists in an initialized output section, the link step must supply raw data to fill it. The link step fills holes with a 32-bit fill value that is replicated through memory until it fills the hole. The link step determines the fill value as follows:

1. If the hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = sign and a 32-bit constant. For example:

```
SECTIONS
{  outsect:
   {
      file1.obj(.text)
      file2.obj(.bss)= 0xFF00FF00 /* Fill this hole with 0xFF00FF00 */
   }
}
```

2. You can also specify a fill value for all the holes in an output section by supplying the fill value after the section definition:

```
SECTIONS
{  outsect:fill = 0xFF00FF00     /* Fills holes with 0xFF00FF00 */
   {
      . += 0x0010;                /* This creates a hole         */
      file1.obj(.text)
      file1.obj(.bss)             /* This creates another hole   */
   }
}
```

3. If you do not specify an initialization value for a hole, the link step fills the hole with the value specified with the --fill_value option (see Section 7.4.5). For example, suppose the command file link.cmd contains the following SECTIONS directive:

```
SECTIONS
{
   .text: { .= 0x0100; }  /* Create a 100 word hole */
}
```

   Now invoke the link step with the --fill_value option:

```
cl2000 -v28 --run_linker --fill_value=0xFFFFFFFF link.cmd
```

   This fills the hole with 0xFFFFFFFF.

4. If you do not invoke the link step with the --fill_value option or otherwise specify a fill value, the link step 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 link step uses to fill it.

7.15.4 Explicit Initialization of Uninitialized Sections

You can force the link step to initialize an uninitialized section by specifying an explicit fill value for it in the SECTIONS directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
   .bss: fill = 0x12341234 /* Fills .bss with 0x12341234 */
}
```

Filling Sections

Note: Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large sections or holes.

7.16 Link-Step-Generated Copy Tables

The link step supports extensions to the link command file syntax that enable the following:

- Make it easier for you to copy objects from load-space to run-space at boot time
- Make it easier for you to manage memory overlays at run time
- Allow you to split GROUPs and output sections that have separate load and run addresses

7.16.1 A Current Boot-Loaded Application Development Process

In some embedded applications, there is a need to copy or download code and/or data from one location to another at boot time before the application actually begins its main execution thread. For example, an application may have its code and/or data in FLASH memory and need to copy it into on-chip memory before the application begins execution.

One way you can develop an application like this is to create a copy table in assembly code that contains three elements for each block of code or data that needs to be moved from FLASH into on-chip memory at boot time:

- The load location (load page id and address)
- The run location (run page id and address)
- The size

The process you follow to develop such an application might look like this:

1. Build the application to produce a .map file that contains the load and run addresses of each section that has a separate load and run placement.
2. Edit the copy table (used by the boot loader) to correct the load and run addresses as well as the size of each block of code or data that needs to be moved at boot time.
3. Build the application again, incorporating the updated copy table.
4. Run the application.

7.16.2 An Alternative Approach

You can avoid some of this maintenance burden by using the LOAD_START(), RUN_START(), and SIZE() operators that are already part of the link command file syntax . For example, instead of building the application to generate a .map file, the link command file can be annotated:

```
SECTIONS
{
   .flashcode: { app_tasks.obj(.text) }
      load = FLASH, run = PMEM,
      LOAD_START(_flash_code_ld_start),
      RUN_START(_flash_code_rn_start),
      SIZE(_flash_code_size)

   ...
}
```

In this example, the LOAD_START(), RUN_START(), and SIZE() operators instruct the link step 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 7.16.1.

While maintenance of the copy table is reduced markedly, you must still carry the burden of keeping the copy table contents in sync with the symbols that are defined in the link command file. Ideally, the link step would generate the boot copy table automatically. This would avoid having to build the application twice *and* free you from having to explicitly manage the contents of the boot copy table.

For more information on the LOAD_START(), RUN_START(), and SIZE() operators, see Section 7.14.7.

7.16.3 Overlay Management Example

Consider an application which contains a memory overlay that must be managed at run time. The memory overlay is defined using a UNION in the link command file as illustrated in Example 7-18:

Example 7-18. Using a UNION for Memory Overlay

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1: { task1.obj(.text) }
         .task2: { task2.obj(.text) }

      } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)

      GROUP
      {
         .task3: { task3.obj(.text) }
         .task4: { task4.obj(.text) }

      } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)

   } run = RAM, RUN_START(_task_run_start)

 ...
}
```

The application must manage the contents of the memory overlay at run time. That is, whenever any services from .task1 or .task2 are needed, the application must first ensure that .task1 and .task2 are resident in the memory overlay. Similarly for .task3 and .task4.

To affect a copy of .task1 and .task2 from ROM to RAM at run time, the application must first gain access to the load address of the tasks (_task12_load_start), the run address (_task_run_start), and the size (_task12_size). Then this information is used to perform the actual code copy.

7.16.4 Generating Copy Tables Automatically With the Link Step

The link step supports extensions to the link command file syntax that enable you to do the following:

- Identify any object components that may need to be copied from load space to run space at some point during the run of an application
- Instruct the link step 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 link step to generate a symbol specified by you that provides the address of a link step-generated copy table. For instance, Example 7-18 can be written as shown in Example 7-19:

Example 7-19. Produce Address for Link Step Generated Copy Table

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1: { task1.obj(.text) }
         .task2: { task2.obj(.text) }

      } load = ROM, table(_task12_copy_table)

      GROUP
      {
         .task3: { task3.obj(.text) }
         .task4: { task4.obj(.text) }

      } load = ROM, table(_task34_copy_table)

   } run = RAM
   ...
}
```

Using the SECTIONS directive from Example 7-19 in the link command file, the link step 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 link step-generated symbols, _task12_copy_table and _task34_copy_table, which provide the addresses of the two copy tables, respectively.

Using this method, you do not have to worry about the creation or maintenance of a copy table. You can reference the address of any copy table generated by the link step in C/C++ or assembly source code, passing that value to a general purpose copy routine which will process the copy table and affect the actual copy.

7.16.5 The table() Operator

You can use the table() operator to instruct the link step 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 link step 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 link step-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 link step detects violations of these rules and reports them as warnings, ignoring each offending use of the table() specification. The link step does not generate a copy table for erroneous table() operator specifications.

7.16.6 Boot-Time Copy Tables

The link step supports a special copy table name, BINIT (or binit), that you can use to create a boot-time copy table. For example, the link command file for the boot-loaded application described in Section 7.16.2 can be rewritten as follows:

```
SECTIONS
{
   .flashcode: { app_tasks.obj(.text) }
   load = FLASH, run = PMEM,
      table(BINIT)
   ...
}
```

For this example, the link step creates a copy table that can be accessed through a special link step-generated symbol, ___binit__, which contains the list of all object components that need to be copied from their load location to their run location at boot-time. If a link command file does not contain any uses of table(BINIT), then the ___binit__ symbol is given a value of -1 to indicate that a boot-time copy table does not exist for a particular application.

You can apply the table(BINIT) specification to an output section, GROUP, or UNION member. If used in the context of a UNION, only one member of the UNION can be designated with table(BINIT). If applied to a GROUP, then none of that GROUP's members may be marked with table(BINIT).The link step detects violations of these rules and reports them as warnings, ignoring each offending use of the table(BINIT) specification.

7.16.7 Using the table() Operator to Manage Object Components

If you have several pieces of code that need to be managed together, then you can apply the same table() operator to several different object components. In addition, if you want to manage a particular object component in multiple ways, you can apply more than one table() operator to it. Consider the link command file excerpt in Example 7-20:

Example 7-20. Link Step Command File to Manage Object Components

```
SECTIONS
{
   UNION
   {
      .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second: { a2.obj(.text), b2.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra: load = EMEM, run = PMEM, table(BINIT)

   ...
}
```

In this example, the output sections .first and .extra are copied from external memory (EMEM) into program memory (PMEM) at boot time while processing the BINIT copy table. After the application has started executing its main thread, it can then manage the contents of the overlay using the two overlay copy tables named: _first_ctbl and _second_ctbl.

7.16.8 Copy Table Contents

In order to use a copy table that is generated by the link step, you must be aware of the contents of the copy table. This information is included in a new run-time-support library header file, cpy_tbl.h, which contains a C source representation of the copy table data structure that is automatically generated by the link step.

Example 7-21 shows the TMS320C28x copy table header file.

Example 7-21. TMS320C28x cpy_tbl.h File

```
/*****************************************************************************/
/* cpy_tbl.h                                                                 */
/*                                                                           */
/* Copyright (c) 2003 Texas Instruments Incorporated                         */
/*                                                                           */
/* Specification of copy table data structures which can be automatically   */
/* generated by the linker (using the table() operator in the LCF).         */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/* Copy Record Data Structure                                                */
/*****************************************************************************/
typedef struct copy_record
{
   unsigned int          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 link step 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 link step collects all COPY_RECORDs that are associated with the same copy table into a COPY_TABLE object. The COPY_TABLE object contains the size of a given COPY_RECORD, the number of COPY_RECORDs in the table, and the array of COPY_RECORDs in the table. For instance, in the BINIT example in Section 7.16.6, the .first and .extra output sections will each have their own COPY_RECORD entries in the BINIT copy table. The BINIT copy table will then look like this:

```
COPY_TABLE __binit__ = { 12, 2,
                         { <load page id of .first>,
                           <run page id of .first>,
                           <load address of .first>,
                           <run address of .first>,
                           <size of .first> },
                         { <load page id of .extra>,
                           <run page id of .extra>,
                           <load address of .extra>,
                           <run address of .extra>,
                           <size of .extra> } };
```

7.16.9 General Purpose Copy Routine

The cpy_tbl.h file in Example 7-21 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 link step-generated copy table. The routine then processes the copy table data object and performs the copy of each object component specified in the copy table.

The copy_in() function definition is provided in the cpy_tbl.c run-time-support source file shown in Example 7-22.

Example 7-22. Run-Time-Support cpy_tbl.c File

```
/*****************************************************************************/
/* cpy_tbl.c                                                                 */
/*                                                                           */
/* Copyright (c) 2003 Texas Instruments Incorporated                         */
/*                                                                           */
/* General purpose copy routine.  Given the address of a linker-generated   */
/* COPY_TABLE data structure, effect the copy of all object components       */
/* that are designated for copy via the corresponding LCF table() operator. */
/*****************************************************************************/
#include <cpy_tbl.h>
#include <string.h>

/*****************************************************************************/
/* COPY_IN()                                                                 */
/*****************************************************************************/
void copy_in(COPY_TABLE *tp)
{
   unsigned int i;
   for (i = 0; i < tp->num_recs; i++)

   {
      COPY_RECORD *crp = &tp->recs[i];
      unsigned int cpy_type = 0;
      unsigned int j;

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

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

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

7.16.10 Link Step Generated Copy Table Sections and Symbols

The link step 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 link step 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 link step creates a single input section, .binit, to contain the entire boot-time copy table.

Example 7-23 illustrates how you can control the placement of the link step-generated copy table sections using the input section names in the link command file.

Example 7-23. Controlling the Placement of the Link-Step-Generated Copy Table Sections

```
SECTIONS
{
   UNION
   {
      .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
              load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second: { a2.obj(.text), b2.obj(.text) }
              load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra: load = EMEM, run = PMEM, table(BINIT)

   ...

   .ovly: { } > BMEM
   .binit: { } > BMEM
}
```

For the link command file in Example 7-23, 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 link step-generated copy table sections, they are allocated according to the link step's default placement algorithm.

The link step does not allow other types of input sections to be combined with a copy table input section in the same output section. The link step does not allow a copy table section that was created from a partial link session to be used as input to a succeeding link session.

7.16.11 Splitting Object Components and Overlay Management

In previous versions of the link step, splitting sections that have separate load and run placement instructions was not permitted. This restriction was because there was no effective mechanism for you, the developer, to gain access to the load address or run address of each one of the pieces of the split object component. Therefore, there was no effective way to write a copy routine that could move the split section from its load location to its run location.

However, the link step 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 link step to generate this information into a copy table. The link step gives each piece of the split object component a COPY_RECORD entry in the copy table object.

For example, consider an application which has 7 tasks. Tasks 1 through 3 are overlaid with tasks 4 through 7 (using a UNION directive). The load placement of all of the tasks is split among 4 different memory areas (LMEM1, LMEM2, LMEM3, and LMEM4). The overlay is defined as part of memory area PMEM. You must move each set of tasks into the overlay at run time before any services from the set are used.

You can use table() operators in combination with splitting operators, >>, to create copy tables that have all the information needed to move either group of tasks into the memory overlay as shown in Example 7-24. Example 7-25 illustrates a possible driver for such an application.

Example 7-24. Creating a Copy Table to Access a Split Object Component

```
SECTIONS
{
   UNION
   {
      .task1to3: { *(.task1), *(.task2), *(.task3) }
              load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)

      GROUP
      {
         .task4: { *(.task4) }
         .task5: { *(.task5) }
         .task6: { *(.task6) }
         .task7: { *(.task7) }

      } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)

   } run = PMEM

   ...

   .ovly: > LMEM4
}
```

Example 7-25. Split Object Component Driver

```
#include <cpy_tbl.h>

extern far COPY_TABLE task13_ctbl;
extern far COPY_TABLE task47_ctbl;

extern void task1(void);
...
extern void task7(void);

main()
{
   ...
   copy_in(&task13_ctbl);
   task1();
   task2();
   task3();
   ...

   copy_in(&task47_ctbl);
   task4();
   task5();
   task6();
   task7();
   ...
}
```

You must declare a COPY_TABLE object as *far* to allow the overlay copy table section placement to be independent from the other sections containing data objects (such as .bss).

The contents of the .task1to3 section are split in the section's load space and contiguous in its run space. The link step-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 link step 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 link step does not permit a split operator to be applied to the run placement of either a UNION or of a UNION member. The link step detects such violations, emits a warning, and ignores the offending split operator usage.

7.17 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 link step *must* have relocation information. Use the --relocatable option when you link the file the first time. (See Section 7.4.1.2.)
- Intermediate files *must* have symbolic information. By default, the link step 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 7.4.12.)
- Intermediate link steps should be concerned only with the formation of output sections and not with allocation. All allocation, binding, and MEMORY directives should be performed in the final link step.
- If the intermediate files have global symbols that have the same name as global symbols in other files and you want them to be treated as static (visible only within the intermediate file), you must link the files with the --make_static option (see Section 7.4.9).
- If you are linking C code, do not use --ram_model or --rom_model until the final link step. Every time you invoke the link step with the --ram_model or --rom_model option, the link step attempts to create an entry point. (See Section 7.4.14.)

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 -v28 --run_linker --relocatable --output_file=tempout1 file1.com
```

file1.com contains:

```
SECTIONS { ss1: { f1.obj f2.obj . . . fn.obj } }
```

Step 2: Link the file file2.com; use the --relocatable option to retain relocation information in the output file tempout2.out.

```
cl2000 -v28 --run_linker --relocatable --output_file=tempout2 file2.com
```

file2.com contains:

```
SECTIONS { ss2: { g1.obj g2.obj . . . gn.obj } }
```

Step 3: Link tempout1.out and tempout2.out.

```
cl2000 -v28 --run_linker --map_file=final.map --output_file=final.out tempout1.out
tempout2.out
```

7.18 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 -v28 --run_linker --rom_model --output_file prog.out prog1.obj prog2.obj ... rts2800.lib
```

The --rom_model option tells the link step 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 C/C++ Compiler User's Guide*

7.18.1 Run-Time Initialization

All C/C++ programs must be linked with code to initialize and execute the program, called a *bootstrap* routine, also known as the *boot.obj* object module. The symbol _c_int00 is defined as the program entry point and is the start of the C boot routine in boot.obj; referencing _c_int00 ensures that boot.obj is automatically linked in from the run-time-support library. When a program begins running, it executes boot.obj first. The boot.obj symbol contains code and data for initializing the run-time environment and performs the following tasks:

- Sets up the system stack and configuration registers
- Processes the run-time *.cinit* initialization table and autoinitializes global variables (when the link step is invoked with the --rom_model option)
- Disables interrupts and calls _main

The run-time-support object libraries contain boot.obj. You can:

- Use the archiver to extract boot.obj from the library and then link the module in directly.
- Include the appropriate run-time-support library as an input file (the link step automatically extracts boot.obj when you use the --ram_model or --rom_model option).

7.18.2 Object Libraries and Run-Time Support

The *TMS320C28x 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 link step includes and links only those library members that resolve undefined references.

7.18.3 Setting the Size of the Stack and Heap Sections

The C/C++ language uses two uninitialized sections called .sysmem and .stack for the memory pool used by the malloc( ) functions and the run-time stacks, respectively. You can set the size of these by using the --heap_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 7.4.6 for setting heap sizes and Section 7.4.17 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.

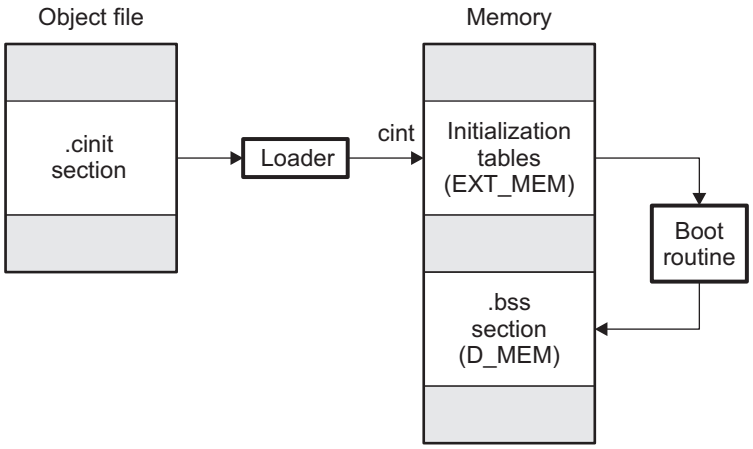
7.18.4 Autoinitialization of Variables at Run Time

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

Using this method, the .cinit section is loaded into memory along with all the other initialized sections. The link step defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in slow external memory and copied to fast external memory each time the program starts.

Figure 7-7 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into slow external memory.

Figure 7-7. Autoinitialization at Run Time



7.18.5 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the link step with the --ram_model option.

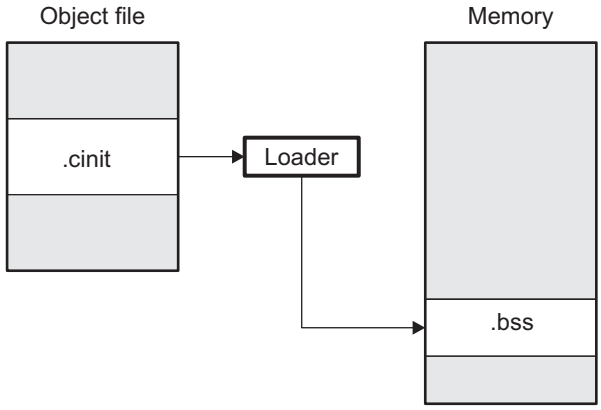
When you use the --ram_model link step option, the link step sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The link step also sets the cinit symbol to -1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

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

- Detect the presence of the .cinit section in the object file.
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory.
- Understand the format of the initialization tables.

Figure 7-8 illustrates the initialization of variables at load time.

Figure 7-8. Initialization at Load Time



7.18.6 The --rom_model and --ram_model Link Step Options

The following list outlines what happens when you invoke the link step with the --ram_model or --rom_model option.

- The symbol _c_int00 is defined as the program entry point. The _c_int00 symbol is the start of the C boot routine in boot.obj; referencing _c_int00 ensures that boot.obj is automatically linked in from the appropriate run-time-support library.
- The .cinit output section is padded with a termination record to designate to the boot routine (autoinitialize at run time) or the loader (initialize at load time) when to stop reading the initialization tables.
- When you initialize at load time (--ram_model option):
  - The link step sets cinit to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag (0010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform initialization directly and not to load the .cinit section into memory. The link step does not allocate space in memory for the .cinit section.
- When you autoinitialize at run time (--rom_model option), the link step defines cinit as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.

Boot Loader

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

7.19 Link Step Example

This example links three object files named demo.obj, ctrl.obj, and tables.obj and creates a program called demo.out.

Assume that target memory has the following program memory configuration:

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.obj, fft.obj, and tables.obj, is linked into program memory ROM.
- Variables, contained in the var_defs section of demo.obj, are linked into data memory in block FAST_MEM_2.
- Tables of coefficients in the .data sections of demo.obj, tables.obj, and fft.obj are linked into FAST_MEM_1. A hole is created with a length of 100 and a fill value of 0x07A1C.
- The xy section form demo.obj, which contains buffers and variables, is linked by default into page 1 of the block STACK, since it is not explicitly linked.

Example 7-26 shows the link command file for this example. Example 7-27 shows the map file.

Example 7-26. Link Step 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.obj
fft.obj
tables.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.obj(.data)              /* .data input */
                fft.obj(.data)                 /* .data input */
                . += 100h;  /* create hole, fill with 0x07A1C */
     }

     var_defs   : { } > FAST_MEM_2 page=1       /* defs in RAM */
     .bss:    page=1, fill=0x0ffff       /*.bss fill and link*/

}

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

Invoke the link step by entering the following command:

```
cl2000 -v28 --run_linker demo.cmd
```

This creates the map file shown in Example 7-27 and an output file called demo.out that can be run on a TMS320C28x.

Example 7-27. 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.obj (.text)
                  003f000e     00000000     tables.obj (.text)
                  003foooe     0000000c     fft.obj (.text)
var_defs    1     003ed000     00000002
                  003ed000     00000002     demo.obj (var_defs)
.data       1     00000200     0000010c
                  00000200     00000004     tables.obj (.data)
                  00000204     00000000     fft.obj (.data)
                  00000204     00000100     --HOLE-- [fill = 7a1c]
                  00000304     00000008     demo.obj (.data)
.bss        0     00000040     00000069
                  00000040     00000068     demo.obj (.bss) [fill=ffff]
                  000000a8     00000000     fft.obj (.bss)
                  000000a8     00000001     tables.obj (.bss) [fill=ffff]
xy          1     000000a9     00000014     UNINITIALIZED
                  000000a9     00000014     demo.obj (xy)

GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

address    name
--------   ----
00000040   .bss
00000200   .data
003f0000   .text
00000040   ARRAY
000000a8   TEMP
00000040   ___bss__
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   ___bss__
00000040   .bss
000000a8   TEMP
000000a9   ___end__
000000a9   end
00000200   ___data__
00000200   .data
0000030c   edata
```

Example 7-27. Output Map File, demo.map (continued)

```
0000030c   ___edata__
003f0000   _main
003f0000   .text
003f0000   ___text__
003f000e   _func1
003f001a   etext
003f001a   ___etext__

[16 symbols]
```

Absolute Lister Description

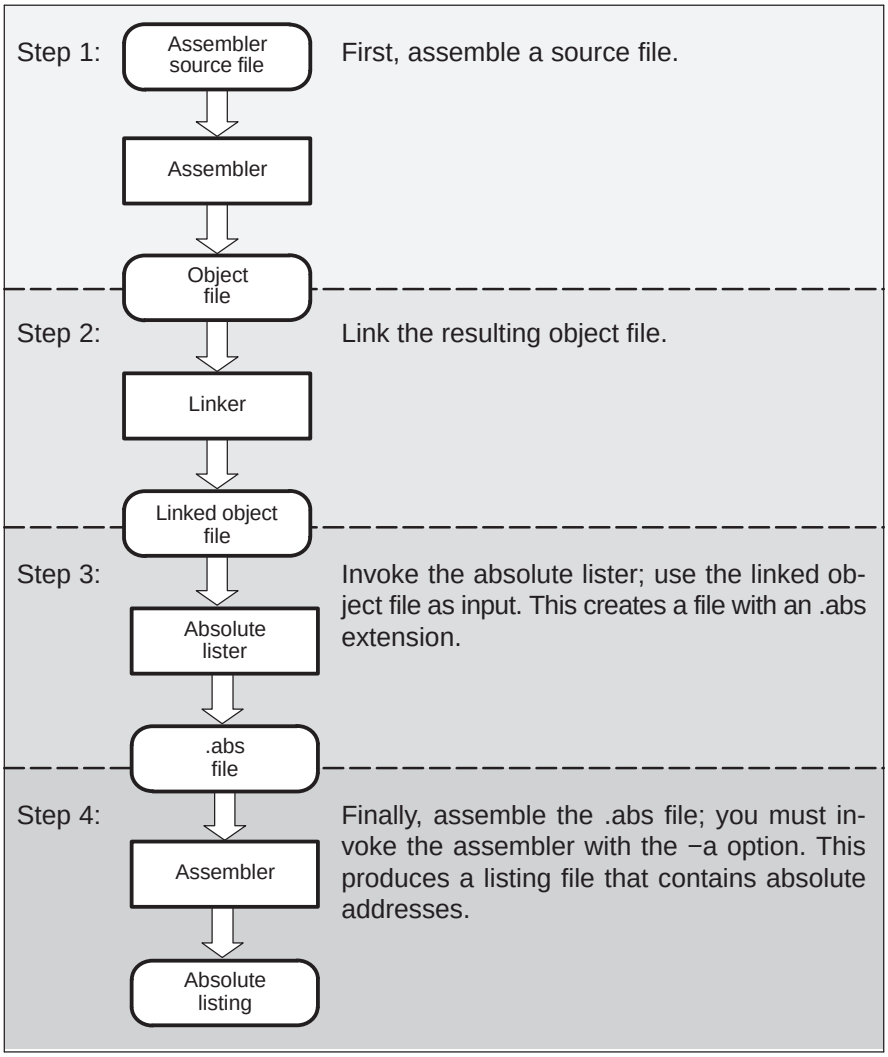
The TMS320C28x™ absolute lister is a debugging tool that accepts linked object files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code. Manually, this could be a tedious process requiring many operations; however, the absolute lister utility performs these operations automatically.

Topic **Page**

8.1 Producing an Absolute Listing

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

Figure 8-1. Absolute Lister Development Flow



8.2 Invoking the Absolute Lister

The syntax for invoking the absolute lister is:

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 three options are listed below.

- ea [.]*asmext* for assembly files (default is .asm)
- ec [.]*cext* for C source files (default is .c)
- eh [.]*hext* for C header files (default is .h)
- 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 -aa assembler option as follows to create the absolute listing:

cl2000 -v28 -aa *filename***.abs**

The -e options affect both the interpretation of filenames on the command line and the names of the output files. They should always precede any filename on the command line.

The -e options are useful when the linked object file was created from C files compiled with the debugging option (-g compiler option). When the debugging option is set, the resulting linked object file contains the name of the source files used to build it. In this case, the absolute lister does not generate a corresponding .abs file for the C header files. Also, the .abs file corresponding to a C source file uses the assembly file generated from the C source file rather than the C source file itself.

For example, suppose the C source file hello.csr is compiled with the debugging option set; the debugging option generates the assembly file hello.s. The hello.csr file includes hello.hsr. Assuming the executable file created is called hello.out, the following command generates the proper .abs file:

```
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.

8.3 Absolute Lister Example

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

module1.asm

```
.text
.bss   array,100
.bss   dflag, 2
.copy  globals.def
MOV     ACC, #offset
MOV     ACC, #dflag
```

module2.asm

```
.bss   offset, 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:

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

   ```
   cl2000 -v28 module1
   cl2000 -v28 module2
   ```

   This creates two object files called module1.obj and module2.obj.

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
      .bss:   >RAM
   }
   ```

   Invoke the linker:

   ```
   cl2000 -v28 -z bttest.cmd
   ```

   This command creates an executable object file called bttest.out; use this new file as input for the absolute lister.

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
.bss      .setsym     000008000h
end       .setsym     000008068h
          .setsect    ".text",000002000h
          .setsect    ".data",000008000h
          .setsect    ".bss",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
.bss      .setsym     000008000h
end       .setsym     000008068h
          .setsect    ".text",000002004h
          .setsect    ".data",000008000h
          .setsect    ".bss",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.

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

```
cl2000 -v28  -aa  module1.abs

cl2000 -v28  -aa  module2.abs
```

This command sequence creates two listing files called module1.lst and module2.lst; no object code is produced. These listing files are similar to normal listing files; however, the addresses shown are absolute addresses.

The absolute listing files created are module1.lst (see Example 8-1) and module2.lst (see Example 8-2).

Example 8-1. module1.lst

```
module1.abs                                                     PAGE    1

     15 002000                .text
     16                       .copy       "module1.asm"
      1 002000               .text
      2 008000               .bss    array,100
      3 008064               .bss    dflag,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 8-2. module2.lst

```
module2.abs                                                     PAGE    1

     15 002004              .text
     16                    .copy       "module2.asm"
      1 008066              .bss    offset,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

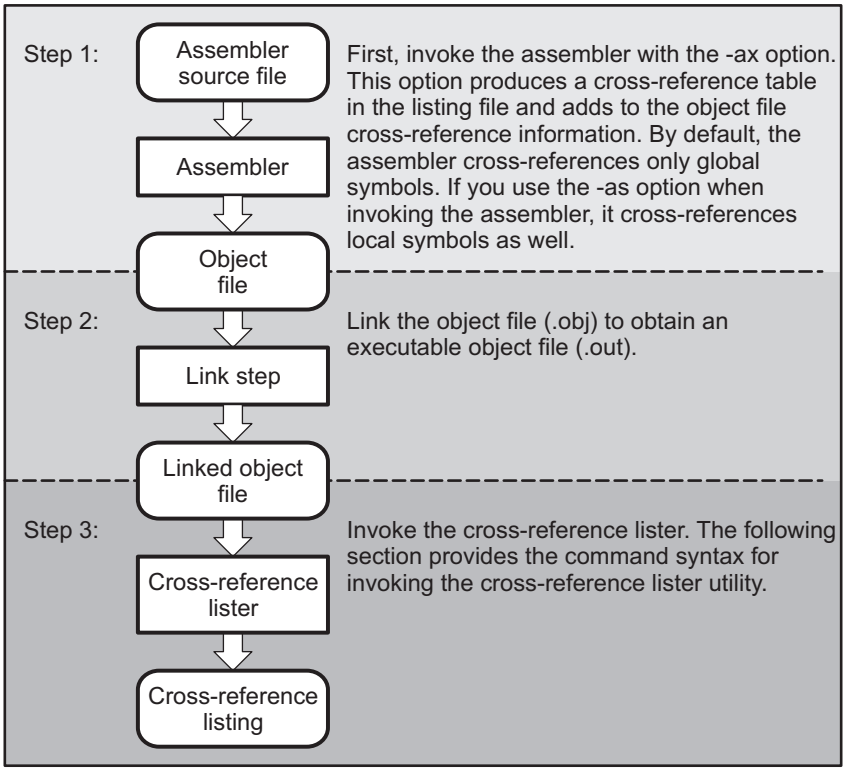
The TMS320C28x™ cross-reference lister is a debugging tool. This utility accepts linked object files as input and produces a cross-reference listing as output. This listing shows symbols, their definitions, and their references in the linked source files.

Topic **Page**

9.1 Producing a Cross-Reference Listing

Figure 9-1 illustrates the steps required to produce a cross-reference listing.

Figure 9-1. The Cross-Reference Lister in the TMS320C28x Software Development Flow



9.2 Invoking the Cross-Reference Lister

To use the cross-reference utility, the file must be assembled with the correct options and then linked into an executable file. Assemble the assembly language files with the -ax option. This option creates a cross-reference listing and adds cross-reference information to the object file. By default, the assembler cross-references only global symbols, but if the assembler is invoked with the -as option, local symbols are also added. Link the object files to obtain an executable file.

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

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.

9.3 Cross-Reference Listing Example

Example 9-1 is an example of cross-reference listing.

Example 9-1. Cross-Reference Listing

```
================================================================================

Symbol: _SETUP

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________   ______  _______  _______  _______
demo.asm         EDEF  '00000018   00000018       18       13       20

================================================================================

Symbol: _fill_tab

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________   ______  _______  _______  _______
ctrl.asm         EDEF  '00000000   00000040       10        5

================================================================================

Symbol: _x42

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________   ______  _______  _______  _______
demo.asm         EDEF  '00000000   00000000        7        4       18

================================================================================

Symbol: gvar

Filename         RTYP    AsmVal     LnkVal      DefLn   RefLn    RefLn    RefLn
________         ____  _________  _________   ______  _______  _______  _______
tables.asm       EDEF  "00000000   08000000       11       10
================================================================================
```

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

Symbol Name of the symbol listed

Filename Name of the file where the symbol appears

RTYP The symbol's reference type in this file. The possible reference types are:

- **STAT** The symbol is defined in this file and is not declared as global.
- **EDEF** The symbol is defined in this file and is declared as global.
- **EREF** The symbol is not defined in this file but is referenced as global.
- **UNDF** The symbol is not defined in this file and is not declared as global.

AsmVal This hexadecimal number is the value assigned to the symbol at assembly time. A value may also be preceded by a character that describes the symbol's attributes. Table 9-1 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 9-1. Symbol Attributes in Cross-Reference Listing

Character	Meaning
'	Symbol defined in a .text section
"	Symbol defined in a .data section
+	Symbol defined in a .sect section
-	Symbol defined in a .bss or .usect section

Object File Utilities Descriptions

This chapter describes how to invoke the following miscellaneous utilities:

- The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both text and XML formats.
- The **name utility** prints a list of names defined and referenced in an object or executable file.
- The **strip utility** removes symbol table and debugging information from object and executable files.

Topic **Page**

10.1 Invoking the Object File Display Utility

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.

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.

--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.

--dynamic_info outputs dynamic linking information for ELF only.

-g appends DWARF debug information to program output.

-h displays help

-o=*filename* sends program output to *filename* rather than to the screen.

--obj_display *attributes* controls the object file display filter settings by specifying a comma-delimited list of *attributes*. When prefixed with no, an attribute is disabled instead of enabled.

Examples:
--obj_display=rawdata,nostrings
--obj_display=all,norawdata
--obj_display=none,header

The ordering of attributes is important. For instance, in "--obj_display=none,header", 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.

-v prints verbose text output.

-x displays output in XML format.

--xml_indent=*num* sets the number of spaces to indent nested XML tags.

If an archive file is given as input to the object file display utility, each object file member of the archive is processed as if it was passed on the command line. The object file members are processed in the order in which they appear in the archive file.

If the object file display utility is invoked without any options, it displays information about the contents of the input files on the console screen.

Object File Display Format

Note: The object file display utility produces data in a text format by default. This data is not intended to be used as machine or software input.

10.2 Invoking the Name Utility

The name utility, *nm2000*, prints the list of names defined and referenced in an object (.obj) or an executable file (.out). It also prints the symbol value and an indication of the kind of symbol.

To invoke the name utility, enter the following:

nm2000 [*-options*] [*input filenames*]

nm2000 is the command that invokes the name utility.

input filename is an object file (.obj) or an executable file (.out).

options identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The name utility options are as follows:

- **-a** prints all symbols.
- **-c** also prints C_NULL symbols for a COFF object module.
- **-d** also prints debug symbols for a COFF object module.
- **-f** prepends file name to each symbol.
- **-g** prints only global symbols.
- **-h** shows the current help screen.
- **-l** produces a detailed listing of the symbol information.
- **-n** sorts symbols numerically rather than alphabetically.
- **-o** *file* outputs to the given file.
- **-p** causes the name utility to not sort any symbols.
- **-q** (quiet mode) suppresses the banner and all progress information.
- **-r** sorts symbols in reverse order.
- **-s** lists symbols in the dynamic symbol table for an ELF object module.
- **-t** also prints tag information symbols for a COFF object module.
- **-u** only prints undefined symbols.

10.3 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:

-o *filename* writes the stripped output to filename.

-p removes all information not required for execution. This option causes more information to be removed than the default behavior, but the object file is left in a state that cannot be linked. This option should be used only with executable (.out) files.

When the strip utility is invoked without the -o option, the input object files are replaced with the stripped version.

Hex Conversion Utility Description

The 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:

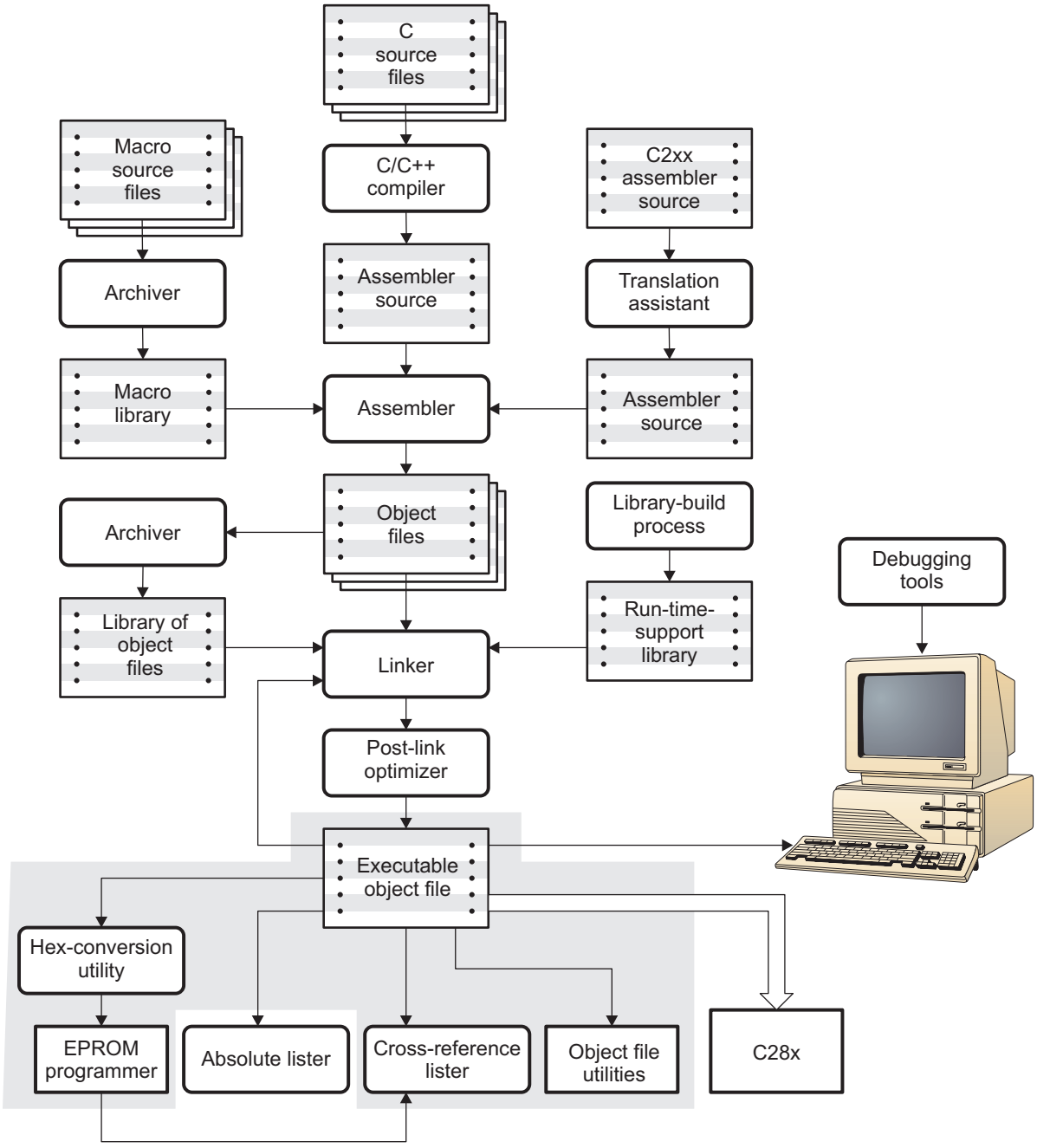
- ASCII-Hex, supporting 16-bit addresses
- Extended Tektronix (Tektronix)
- Intel MCS-86 (Intel)
- Motorola Exorciser (Motorola-S), supporting 16-bit addresses
- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses

Topic **Page**

11.1 The Hex Conversion Utility's Role in the Software Development Flow

Figure 11-1 highlights the role of the hex conversion utility in the software development process.

Figure 11-1. The Hex Conversion Utility in the TMS320C28x Software Development Flow



11.2 Invoking the Hex Conversion Utility

There are two basic methods for invoking the hex conversion utility:

- **Specify the options and filenames on the command line.** The following example converts the file firmware.out into TI-Tagged format, producing two output files, firm.lsb and firm.msb.

```
hex2000 -t firmware -o firm.lsb -o firm.msb
```

- **Specify the options and filenames in a command file.** You can create a batch file that stores command line options and filenames for invoking the hex conversion utility. The following example invokes the utility using a command file called hexutil.cmd:

```
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.

11.2.1 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 11-1 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 multicharacter names must be spelled exactly as shown in this document; no abbreviations are allowed.
- Options are not affected by the order in which they are used. The exception to this rule is the -q (quiet) option, which must be used before any other options.

filename names an object file or a command file (for more information, see Section 11.2.2). If you do not specify a filename, the utility prompts you for one.

Table 11-1. Basic Hex Conversion Utility Options

General Options	Option	Description	See
Control the overall operation of the hex conversion utility.	-exclude=*section_name*	Ignore specified section	Section 11.6
	-linkerfill	Include linker fill sections in images	
	-map=*filename*	Generate a map file	Section 11.4.2
	-o=*filename*	Specify an output filename	Section 11.7
	-quiet, -q	Run quietly (when used, it must appear *before* other options)	Section 11.2.2
Image Options	**Option**	**Description**	**See**
Create a continuous image of a range of target memory	-fill=*value*	Fill holes with *value*	Section 11.8.2
	-image	Specify image mode	Section 11.8.1
	-zero	Reset the address origin to 0 in image mode	Section 11.8.3
Memory Options	**Option**	**Description**	**See**
Configure the memory widths for your output files	-memwidth=*value*	Define the system memory word width (default 32 bits)	Section 11.3.2
	-romwidth=*value*	Specify the ROM device width (default depends on format used)	Section 11.3.3
	-order LS	Output file is in little-endian format	Section 11.3.4
	-order MS	Output file is in big-endian format	Section 11.3.4
Output Options	**Option**	**Description**	**See**
Specify the output format	-a	Select ASCII-Hex	Section 11.11.1
	-I	Select Intel	
	-m	Select Motorola-S	Section 11.11.3
	-t	Select TI-Tagged	Section 11.11.4
	-ti_txt	Select TI-Txt	
	-x	Select Tektronix (default)	Section 11.11.5
Boot Options	**Option**	**Description**	**See**
Control the boot loader	-boot	Convert all sections into bootable form (use instead of a SECTIONS directive)	
	-bootorg=*addr*	Specify the source address of the boot loader table	
	-e	Specify the entry point at which to begin execution after boot loading	
	-gpio8	Specify table source as the GP I/O port, 8-bit mode. (Aliased by --can8).	
	-gpio16	Specify table source as the GP I/O port, 16-bit mode	
	-lospcp=*value*	Specify the initial value for the LOSPCP register	
	-sci8	Specify table source as the SCI-A port, 8-bit mode	
	-spi8	Specify table source as the SPI-A port, 8-bit mode	
	-spibrr=*value*	Specify the initial value for the SPIBRR register	

11.2.2 Invoking the Hex Conversion Utility With a Command File

A command file is useful if you plan to invoke the utility more than once with the same input files and options. It is also useful if you want to use the ROMS and SECTIONS hex conversion utility directives to customize the conversion process.

Command files are ASCII files that contain one or more of the following:

- **Options and filenames.** These are specified in a command file in exactly the same manner as on the command line.
- **ROMS directive.** The ROMS directive defines the physical memory configuration of your system as a list of address-range parameters. (See Section 11.4.)
- **SECTIONS directive.** The hex conversion utility SECTIONS directive specifies which sections from the object file are selected. (See Section 11.5.)
- **Comments.** You can add comments to your command file by using the /* and */ delimiters. For example:

```
/*   This is a comment.   */
```

To invoke the utility and use the options you defined in a command file, enter:

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 **-q option** suppresses the hex conversion utility's normal banner and progress information.

- Assume that a command file named firmware.cmd contains these lines:

```
firmware.out   /* input file  */
-t             /* TI-Tagged   */
-o   firm.lsb  /* output file */
-o   firm.msb  /* output file */
```

  You can invoke the hex conversion utility by entering:

```
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   */
-I               /* 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, .const:
}
```

11.3 Understanding Memory Widths

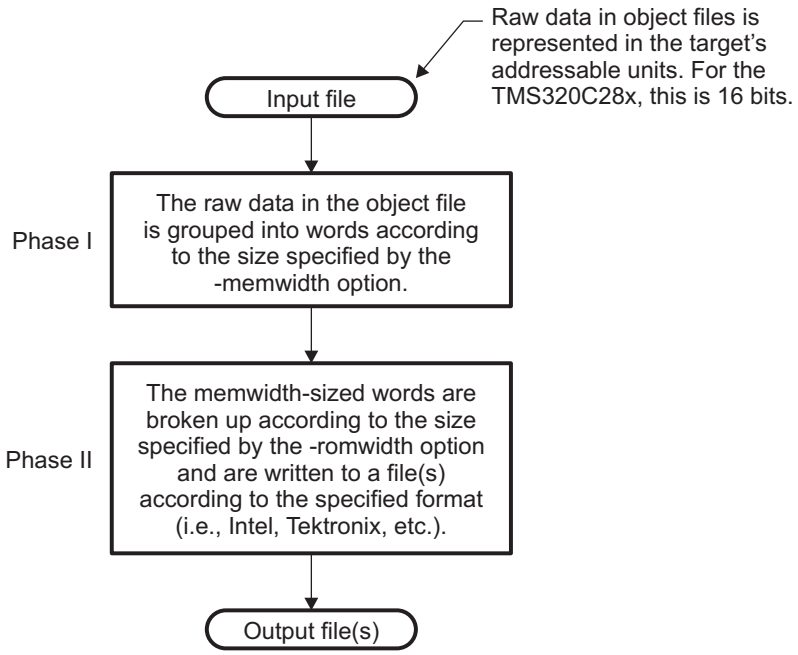
The hex conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex conversion utility, you must understand how the utility treats word widths. Three widths are important in the conversion process:

- Target width
- Memory width
- ROM width

The terms target word, memory word, and ROM word refer to a word of such a width.

Figure 11-2 illustrates the two separate and distinct phases of the hex conversion utility's process flow.

Figure 11-2. Hex Conversion Utility Process Flow



11.3.1 Target Width

Target width is the unit size (in bits) of the target processor's word. The unit size corresponds to the data bus size on the target processor. The width is fixed for each target and cannot be changed. The TMS320C28x targets have a width of 16 bits.

11.3.2 Specifying the Memory Width

Memory width is the physical width (in bits) of the memory system. Usually, the memory system is physically the same width as the target processor width: a 32-bit processor has a 32-bit memory architecture. However, some applications require target words to be broken into multiple, consecutive, and narrower memory words.

By default, the hex conversion utility sets memory width to the target width (in this case, 16 bits).

You can change the memory width by:

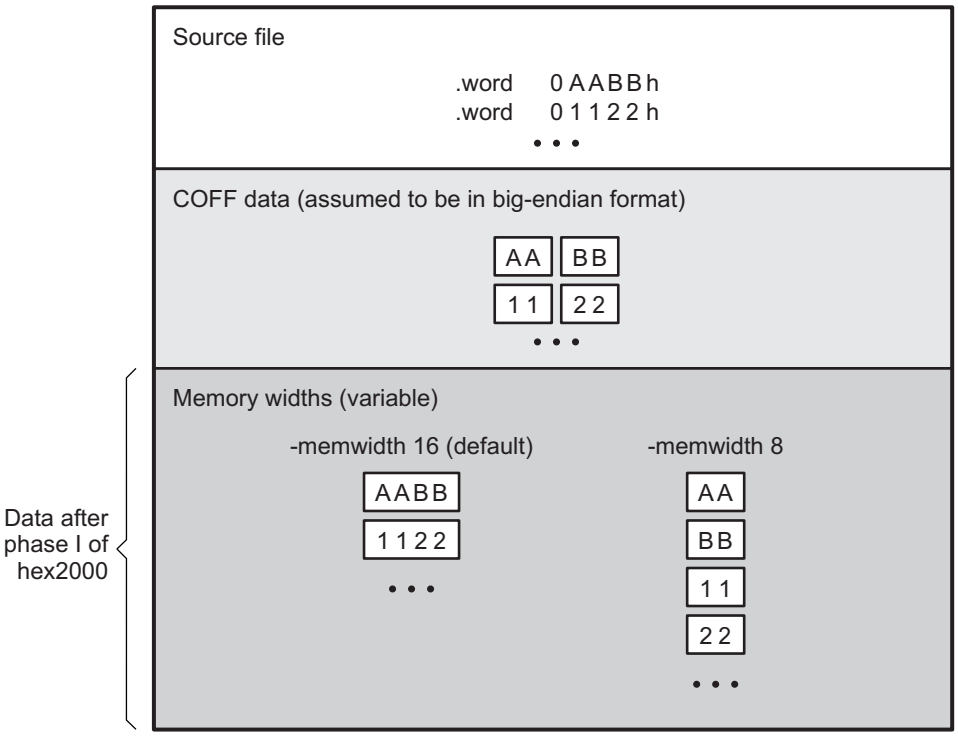
- Using the **-memwidth** option. This changes the memory width value for the entire file.
- Setting the **memwidth** parameter of the ROMS directive. This changes the memory width value for the address range specified in the ROMS directive and overrides the -memwidth option for that range. See Section 11.4.

For both methods, use a value that is a power of 2 greater than or equal to 8.

You should change the memory width default value of 16 only when you need to break single target words into consecutive, narrower memory words.

Figure 11-3 demonstrates how the memory width is related to object file data.

Figure 11-3. Object File Data and Memory Widths



11.3.3 Partitioning Data Into Output Files

ROM width specifies the physical width (in bits) of each ROM device and corresponding output file (usually one byte or eight bits). The ROM width determines how the hex conversion utility partitions the data into output files. After the 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 bits 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.

You can change ROM width (except for TI-Tagged format) by:

- Using the **-romwidth** option. This option changes the ROM width value for the entire 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 11.4.

For both methods, use a value that is a power of 2 greater than or equal to 8.

If you select a ROM width that is wider than the natural size of the output format (16 bits for TI-Tagged or 8 bits for all others), the utility simply writes multibyte fields into the file.

Figure 11-4 illustrates how the 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:

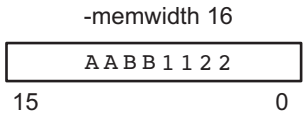
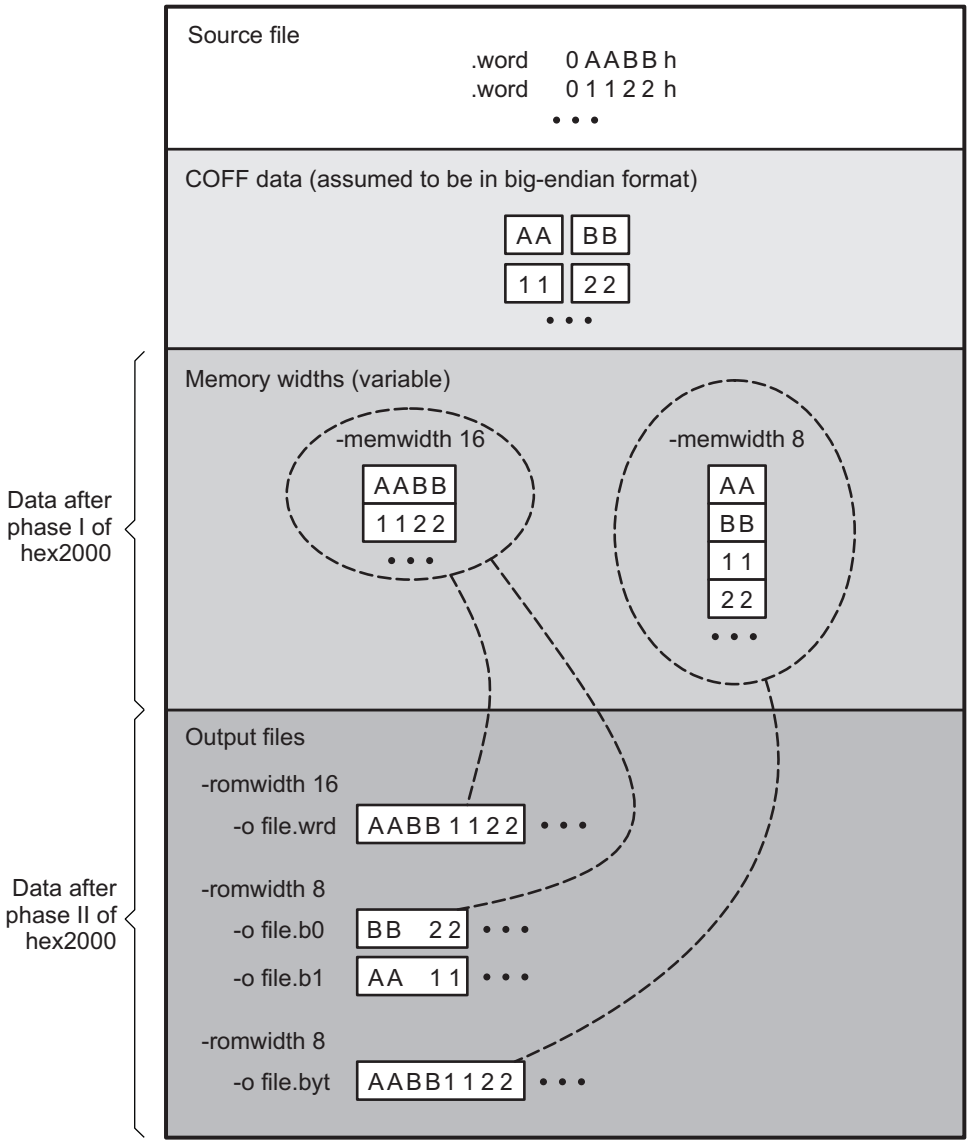


Figure 11-4. Data, Memory, and ROM Widths



11.3.4 Specifying Word Order for Output Words

There are two ways to split a wide word into consecutive memory locations in the same hex conversion utility output file:

- **-order 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.

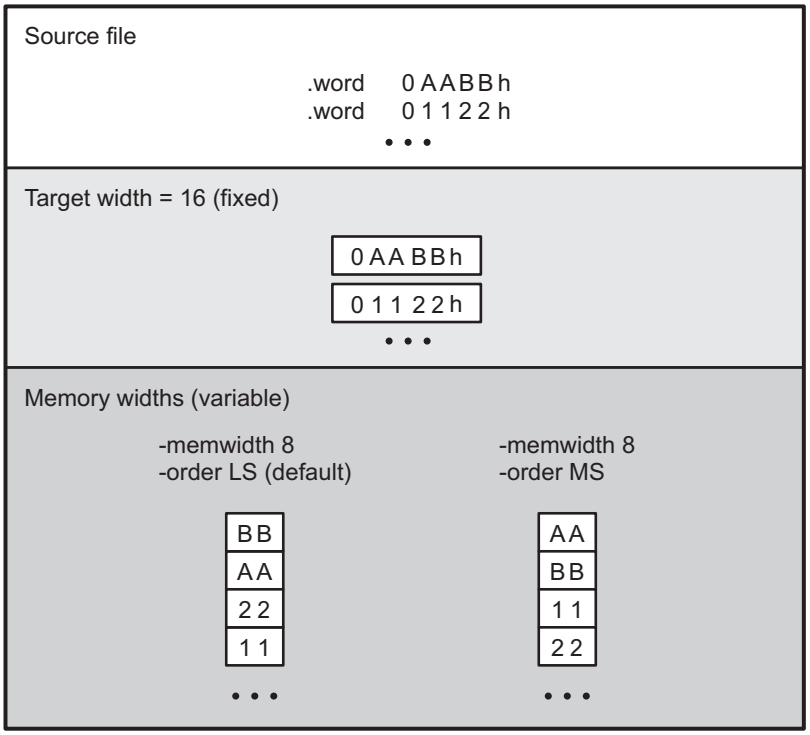
When the -order Option Applies

Notes:

- This option applies only when you use a memory width with a value less than 16. Otherwise, -order is ignored.
- This option does not affect the way memory words are split into output files. Think of the files as a set: the set contains a least significant file and a most significant file, but there is no ordering over the set. When you list filenames for a set of files, you always list the least significant first, regardless of the -order option.

Figure 11-5 demonstrates how the -order option affects the conversion process. This figure and Figure 11-4 explain the condition of the data in the hex conversion utility output files.

Figure 11-5. Varying the Word Order



11.4 The ROMS Directive

The ROMS directive specifies the physical memory configuration of your system as a list of address-range parameters.

Each address range produces one set of files containing the hex conversion utility output data that corresponds to that address range. Each file can be used to program one single ROM device.

The ROMS directive is similar to the MEMORY directive of the TMS320C28x link step: 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. (Duplicate memory range names are allowed.)

origin specifies the starting address of a memory range. It can be entered as origin, org, or o. The associated value must be a decimal, octal, or hexadecimal constant. If you omit the origin value, the origin defaults to 0. The following table summarizes the notation you can use to specify a decimal, octal, or hexadecimal constant:

Constant	Notation	Example
Hexadecimal	0x prefix or h suffix	0x77 or 077h
Octal	0 prefix	077
Decimal	No prefix or suffix	77

length specifies the length of a memory range as the physical length of the ROM device. It can be entered as length, len, or l. The value must be a decimal, octal, or hexadecimal constant. If you omit the length value, it defaults to the length of the entire address space.

romwidth specifies the physical ROM width of the range in bits (see Section 11.3.3). 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 11.3.2). 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 11.5.)

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 11.8.2.)

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 11.3.3. 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.

11.4.1 When to Use the ROMS Directive

If you do not use a ROMS directive, the utility defines a single default range that includes the entire address space. This is equivalent to a ROMS directive with a single range without origin or length.

Use the ROMS directive when you want to:

- **Program large amounts of data into fixed-size ROMs**. When you specify memory ranges corresponding to the length of your ROMs, the utility automatically breaks the output into blocks that fit into the ROMs.
- **Restrict output to certain segments**. You can also use the ROMS directive to restrict the conversion to a certain segment or segments of the target address space. The utility does not convert the data that falls outside of the ranges defined by the ROMS directive. Sections can span range boundaries; the utility splits them at the boundary into multiple ranges. If a section falls completely outside any of the ranges you define, the utility does not convert that section and issues no messages or warnings. Thus, you can exclude sections without listing them by name with the SECTIONS directive. However, if a section falls partially in a range and partially in unconfigured memory, the utility issues a warning and converts only the part within the range.
- **Use image mode.** When you use the -image option, you must use a ROMS directive. Each range is filled completely so that each output file in a range contains data for the whole range. Holes before, between, or after sections are filled with the fill value from the ROMS directive, with the value specified with the -fill option, or with the default value of 0.

11.4.2 An Example of the ROMS Directive

The ROMS directive in Example 11-1 shows how 16K bytes of 16-bit memory could be partitioned for two 8K-byte 8-bit EPROMs. Figure 11-6 illustrates the input and output files.

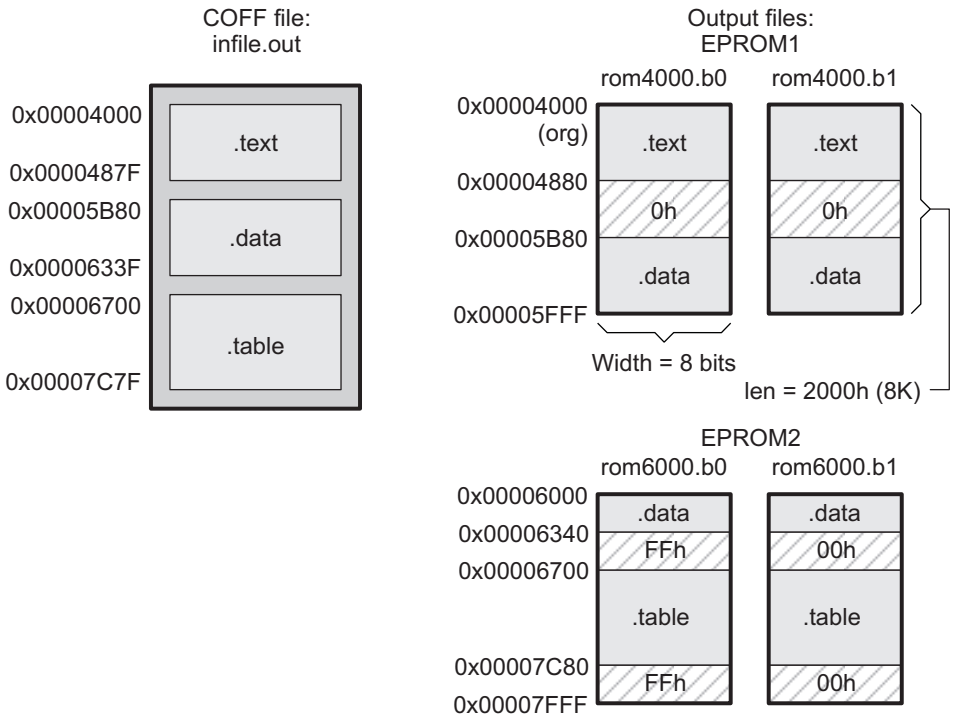
Example 11-1. A ROMS Directive Example

```
infile.out
-image
-memwidth 16

ROMS
}
   EPROM1: org = 0x00004000, len = 0x2000, romwidth = 8
           files = { rom4000.b0, rom4000.b1}

   EPROM2: org = 0x00006000, len = 0x2000, romwidth = 8,
           fill = 0xFF00FF00,
```

Example 11-1. A ROMS Directive Example (continued)

```
            files = { rom6000.b0, rom6000.b1}
}
```

Figure 11-6. The infile.out File Partitioned Into Four Output Files



The map file (specified with the -map option) is advantageous when you use the ROMS directive with multiple ranges. The map file shows each range, its parameters, names of associated output files, and a list of contents (section names and fill values) broken down by address. Example 11-2 is a segment of the map file resulting from the example in Example 11-1.

Example 11-2. Map File Output From Example 11-1 Showing Memory Ranges

```
-------------------------------------------------------
00004000..00005fff Page=0 Width=8 "EPROM1"
-------------------------------------------------------
   OUTPUT FILES:   rom4000.b0   [b0..b7]
                   rom4000.b1   [b8..b15]
   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
-------------------------------------------------------
00006000..00007fff Page=0 Width=8 "EPROM2"
-------------------------------------------------------
   OUTPUT FILES:   rom6000.b0   [b0..b7]
                   rom6000.b1   [b8..b15]
   CONTENTS: 00006000..0000633f .data
             00006340..000066ff FILL = ff00ff00
             00006700..00007c7f .table
             00007c80..00007fff FILL = ff00ff00
```

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 Where is hex_exfill_value (from the specified fill value). The data from this range is converted into two output files:

- rom6000.b0 contains bits 0 through 7
- rom6000.b1 contains bits 8 through 15

11.5 The SECTIONS Directive

You can convert specific sections of the 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. For the TMS320C28x these sections are .text, .const, and .cinit.

Uninitialized sections are *never* converted, whether or not you specify them in a SECTIONS directive.

Sections Generated by the C/C++ Compiler

Note: The TMS320C28x C/C++ compiler automatically generates these sections:

- **Initialized sections**: .text, .const, and .cinit.
- **Uninitialized sections:** .bss, .stack, and .sysmem

Use the SECTIONS directive in a command file. (See Section 11.2.2.) The general syntax for the SECTIONS directive is:

SECTIONS
{
sname[**:**] [**paddr**=*value*]
sname[**:**] [**paddr**=**boot**]
sname[**:**] [**boot**]
...
}

SECTIONS	begins the directive definition.
sname	identifies a section in the input file. If you specify a section that does not exist, the utility issues a warning and ignores the name.
paddr=*value*	specifies the physical ROM address at which this section should be located. This value overrides the section load address given by the linker. This value must be a decimal, octal, or hexadecimal constant. It can also be the word **boot** (to indicate a boot table section for use with a boot loader). *If your file contains multiple sections, and if one section uses a paddr parameter, then all sections must use a paddr parameter.*
boot	configures a section for loading by a boot loader. This is equivalent to using **paddr=boot**. Boot sections have a physical address determined by the location of the boot table. The origin of the boot table is specified with the -bootorg option.

For more similarity with the linker's SECTIONS directive, you can use colons after the section names (in place of the equal sign on the boot keyboard). For example, the following statements are equivalent:

```
SECTIONS { .text: .data: boot }
SECTIONS { .text: .data = boot }
```

In the example below, the object file contains six initialized sections: .text, .data, .const, .vectors, .coeff, and .tables. Suppose you want only .text and .data to be converted. Use a SECTIONS directive to specify this:

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

To configure both of these sections for boot loading, add the boot keyword:

```
SECTIONS { .text = boot .data = boot }
```

For more information about -boot and other command line options associated with boot tables, see Section 11.2.

11.6 Excluding a Specified Section

The -exclude *section_name* option can be used to inform the hex utility to ignore the specified section. If a SECTIONS directive is used, it overrides the -exclude option.

For example, if a SECTIONS directive containing the section name *mysect* is used and an -exclude *mysect* is specified, the SECTIONS directive takes precedence and *mysect* is not excluded.

The -exclude option has a limited wildcard capability. The * character can be placed at the beginning or end of the name specifier to indicate a suffix or prefix, respectively. For example, -exclude sect* disqualifies all sections that begin with the characters sect.

If you specify the -exclude option on the command line with the * wildcard, enter quotes around the section name and wildcard. For example, -exclude"sect*". Using quotes prevents the * form being interpreted by the hex conversion utility. If -exclude is in a command file, then the quotes should not be specified.

11.7 Assigning Output Filenames

When the hex conversion utility translates your 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 -o options.** You can specify names for the output files by using the -o option. If no filenames are listed in the ROMS directive and you use -o options, the utility takes the filename from the list of -o options. The following line has the same effect as the example above using the ROMS directive:

   ```
   -o xyz.b0 -o xyz.b1
   ```

   If both the ROMS directive and -o options are used together, the ROMS directive overrides the -o options.
3. **It assigns a default filename.** If you specify no filenames or fewer names than output files, the utility assigns a default filename. A default filename consists of the base name from the input file plus a 2- to 3-character extension. The extension has three parts:

   a. A format character, based on the output format (see Section 11.11):

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.

   If you include the following ROMS directive when you invoke the hex conversion utility, you would have four output files:

   ```
   ROMS
   {
      range1: o = 0x00001000 l = 0x1000
      range2: o = 0x00002000 l = 0x1000
   }
   ```

These output files ...	Contain data in these locations ...
a.i00 and a.i01	0x00001000 through 0x00001FFF
a.i10 and a.i11	0x00002000 through 0x00002FFF

11.8 Image Mode and the -fill Option

This section points out the advantages of operating in image mode and describes how to produce output files with a precise, continuous image of a target memory range.

11.8.1 Generating a Memory Image

With the -image option, the utility generates a memory image by completely filling all of the mapped ranges specified in the ROMS directive.

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.

11.8.2 Specifying a Fill Value

The -fill option specifies a value for filling the holes between sections. The fill value must be specified as an integer constant following the -fill option. The width of the constant is assumed to be that of a word on the target processor. For example, specifying -fill 0FFh results in a fill pattern of 0FFh.. The constant value is not sign extended.

The hex conversion utility uses a default fill value of 0 if you do not specify a value with the fill option. *The -fill option is valid only when you use -image*; otherwise, it is ignored.

11.8.3 Steps to Follow in Using Image Mode

Step 1: Define the ranges of target memory with a ROMS directive. See Section 11.4.

Step 2: Invoke the hex conversion utility with the -image option. You can optionally use the -zero option to reset the address origin to 0 for each output file. If you do not specify a fill value with the ROMS directive and you want a value other than the default of 0, use the -fill option.

11.9 Building a Table for an On-Chip Boot Loader

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.

11.9.1 Description of the Boot Table

The input for a boot loader is the boot table. The boot table contains records that instruct the 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.

11.9.2 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.

11.9.3 How to Build the Boot Table

Table 11-2 summarizes the hex conversion utility options available for the boot loader.

Table 11-2. Boot-Loader Options

Option	Description
-boot	Convert all sections into bootable form (use instead of a SECTIONS directive).
-bootorg=*value*	Specify the source address of the boot-loader table.
-e *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.

11.9.3.1 Building the Boot Table

To build the boot table, follow these steps:

Step 1: **Link the file.** Each block of the boot table data corresponds to an initialized section in the object file. Uninitialized sections are not converted by the hex conversion utility (see Section 11.5).

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 11.5). 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 11.3 and Section 11.4.

11.9.3.2 Leaving Room for the Boot Table

The complete boot table is similar to a single section containing all of the header records and data for the boot loader. The address of this section is the boot table origin. As part of the normal conversion process, the hex conversion utility converts the boot table to hexadecimal format and maps it into the output files like any other section.

Be sure to leave room in your system memory for the boot table, especially when you are using the ROMS directive. The boot table cannot overlap other nonboot sections or unconfigured memory. Usually, this is not a problem; typically, a portion of memory in your system is reserved for the boot table. Simply configure this memory as one or more ranges in the ROMS directive, and use the -bootorg option to specify the starting address.

11.9.4 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.

11.9.5 Setting the Entry Point for the Boot Table

After completing the boot load process, execution starts at the default entry point specified by the link step and contained in the object file. By using the -e option with the hex conversion utility, you can set the entry point to a different address.

For example, if you want your program to start running at address 0x0123 after loading, specify -e=0x0123 on the command line or in a command file. You can determine the -e address by looking at the map file that the link step generates.

Valid Entry Points

Note: The value can be a constant, or it can be a symbol that is externally defined (for example, with a .global) in the assembly source.

11.9.6 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 11-3.

Table 11-3. 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 11-4

Table 11-4. 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	Blcok 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 -spi8 when invoking the utility. Do not specify -memwidth or -romwidth.

- To boot from a SPI-A port, specify -sci8 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.

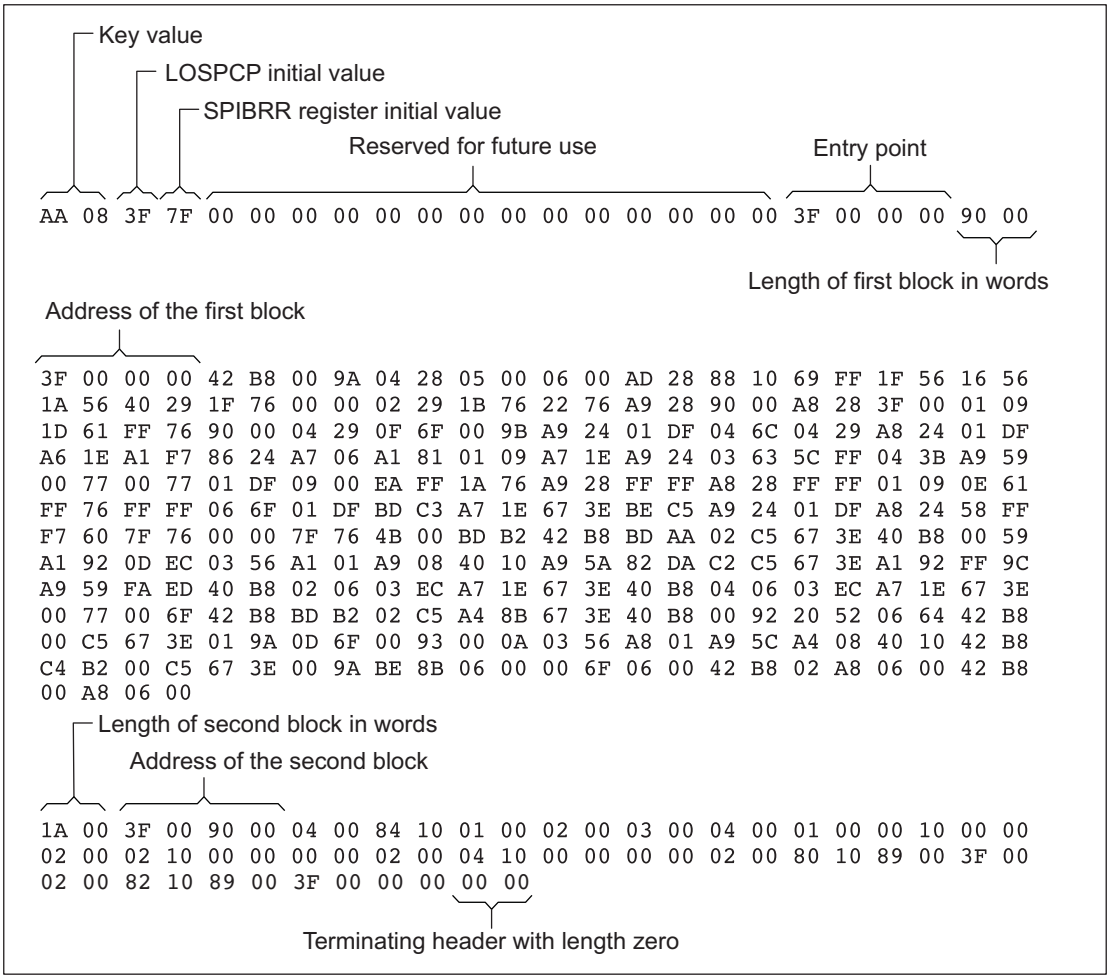
The command file in Example 11-3 allows you to boot the .text and .cinit sections of test.out from a 16-bit-wide EPROM at location 0x3FFC00. The map file test.map is also generated.

Example 11-3. Sample Command File for Booting From 8-Bit SPI Boot

```
/*---------------------------------------------------------------------*/
/* Hex converter command file.                                         */
/*---------------------------------------------------------------------*/
test.out           /* Input COFF file */
-a                 /* Select ASCII format */
-map=test.map      /* Specify the map file */
-o=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) */
-e=0x3F0000        /* Set the entry point */
```

The command file in Example 11-3 generates the out file in Figure 11-7. The control register values are coded in the boot table header and that header has the address that is specified with the -e option.

Figure 11-7. Sample Hex Converter Out File for Booting From 8-Bit SPI Boot



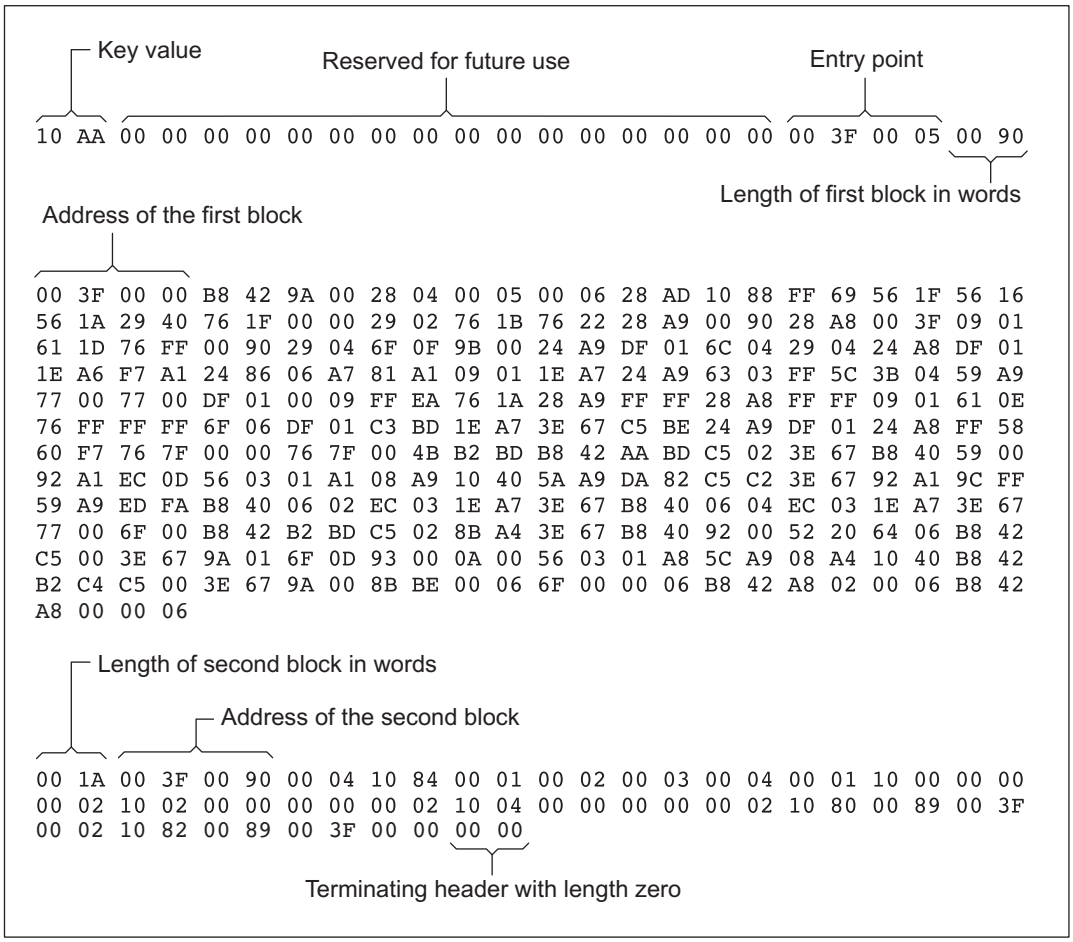
The command file in Example 11-4 allows you to boot the .text and .cinit sections of test.out from the 16-bit parallel GP I/O port. The map file test.map is also generated.

Example 11-4. Sample Command File for C28x 16-Bit Parallel Boot GP I/O

```
/*--------------------------------------------------------------------*/
/* Hex converter command file.                                        */
/*--------------------------------------------------------------------*/
test.out            /* Input COFF file */
-a                  /* Select ASCII format */
-map=test.map       /* Specify the map file */
-o=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 11-4 generates the out file in Figure 11-8.

Figure 11-8. Sample Hex Converter Out File for C28x 16-Bit Parallel Boot GP I/O



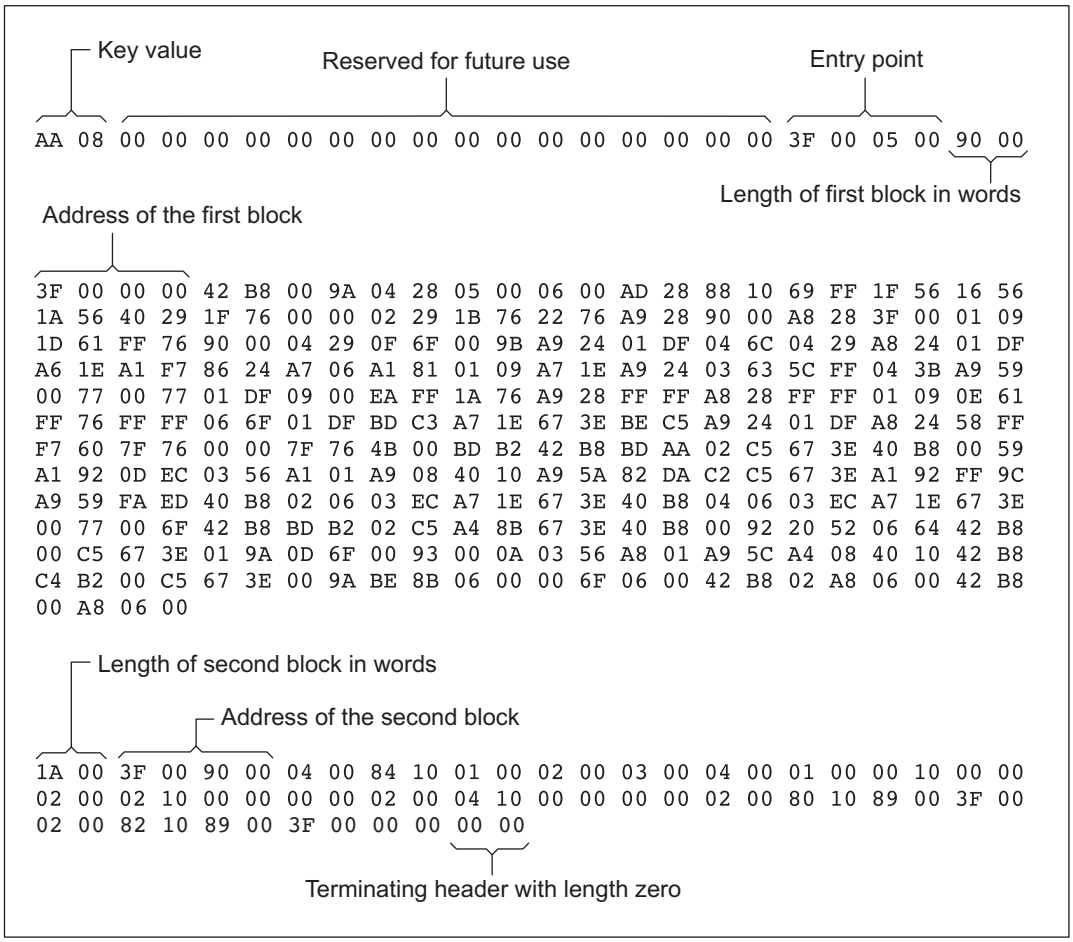
The command file in Section 11.9.6.1 allows you to boot the .text and .cinit sections of test.out from a 16-bit wide EPROM from the SCI-A 8-bit port. The map file test.map is also generated.

11.9.6.1 Sample Command File for Booting From 8-Bit SCI Boot

```
/*--------------------------------------------------------------------*/
/* Hex converter command file.                                        */
/*--------------------------------------------------------------------*/
test.out            /* Input COFF file */
-a                  /* Select ASCII format */
-map=test.map       /* Specify the map file */
-o=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 Section 11.9.6.1 generates the out file in Figure 11-9.

Figure 11-9. Sample Hex Converter Out File for Booting From 8-Bit SCI Boot



11.10 Controlling the ROM Device Address

The hex conversion utility output address field corresponds to the ROM device address. The EPROM programmer burns the data into the location specified by the hex conversion utility output file address field. The hex conversion utility offers some mechanisms to control the starting address in ROM of each section. However, many EPROM programmers offer direct control of the location in ROM in which the data is burned.

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.

11.11 Description of the Object Formats

The hex conversion utility has options that identify each format. Table 11-5 specifies the format options. They are described in the following sections.

- You need to use only one of these options on the command line. If you use more than one option, the last one you list overrides the others.
- The default format is Tektronix (-x option).

Table 11-5. Options for Specifying Hex Conversion Formats

Option	Format	Address Bits	Default Width
-a	ASCII-Hex	16	8
-I	Intel	32	8
-m	Motorola-S	32	8
-t	TI-Tagged	16	16
-x	Tektronix	32	8

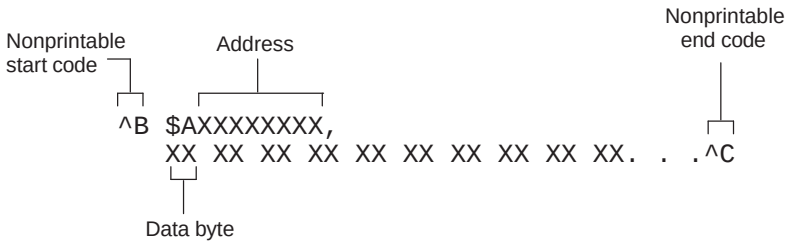
Address bits determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

The **default width** determines the default output width of the format. You can change the default width by using the -romwidth option or by using the romwidth parameter in the ROMS directive. You cannot change the default width of the TI-Tagged format, which supports a 16-bit width only.

11.11.1 ASCII-Hex Object Format (-a Option)

The ASCII-Hex object format supports 16-bit addresses. The format consists of a byte stream with bytes separated by spaces. Figure 11-10 illustrates the ASCII-Hex format.

Figure 11-10. ASCII-Hex Object Format



The file begins with an ASCII STX character (ctrl-B, 02h) and ends with an ASCII ETX character (ctrl-C, 03h). Address records are indicated with $AXXXXXXX, in which XXXXXXX 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.

11.11.2 Intel MCS-86 Object Format (-I Option)

The Intel object format supports 16-bit addresses and 32-bit extended addresses. Intel format consists of a 9-character (4-field) prefix (which defines the start of record, byte count, load address, and record type), the data, and a 2-character checksum suffix.

The 9-character prefix represents three record types:

Record Type	Description
00	Data record
01	End-of-file record
04	Extended linear address record

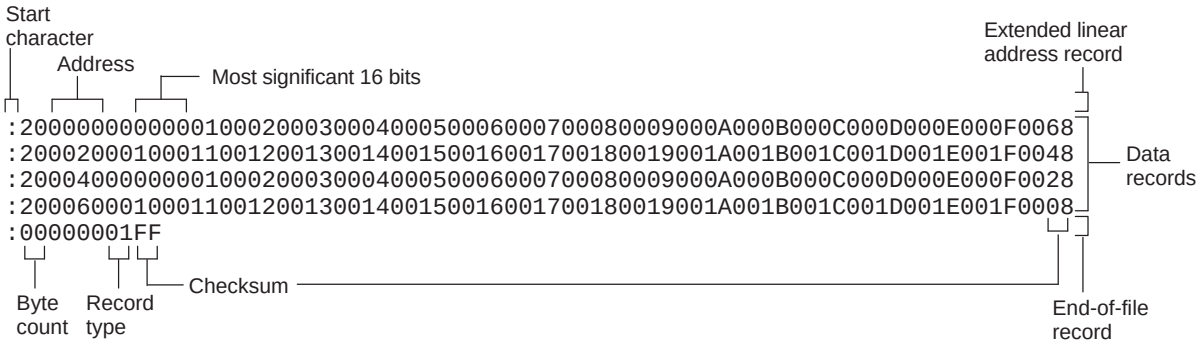
Record type00, the data record, begins with a colon (:) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended 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 11-11 illustrates the Intel hexadecimal object format.

Figure 11-11. Intel Hexadecimal Object Format



11.11.3 Motorola Exorciser Object Format (-m Option)

The Motorola-S format supports 32-bit addresses. It consists 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:

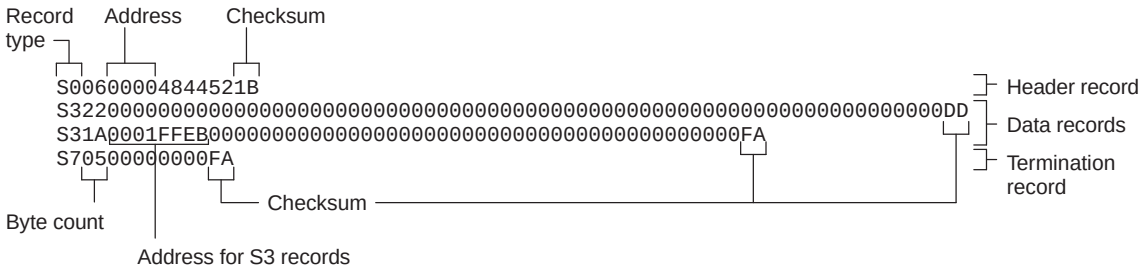
Record Type	Description
S0	Header record
S3	Code/data record
S7	Termination record

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 11-12 illustrates the Motorola-S object format.

Figure 11-12. Motorola-S Format



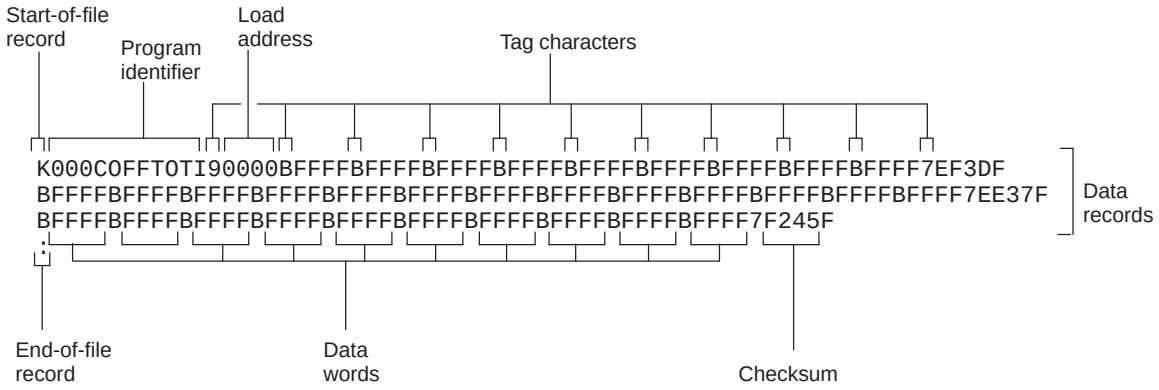
11.11.4 Texas Instruments SDSMAC Object Format (-t Option)

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 11-13 illustrates the tag characters and fields in TI-Tagged object format.

Figure 11-13. 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 (:).

11.11.5 Extended Tektronix Object Format (-x 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.

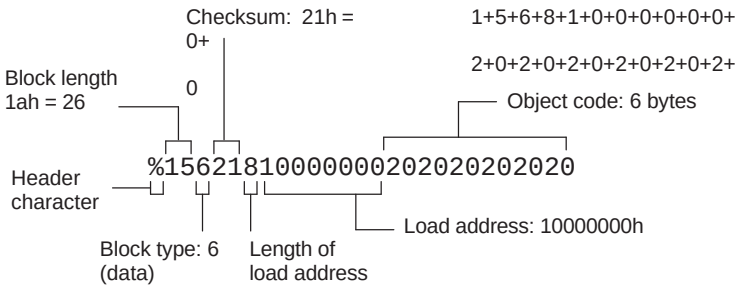
The header field in the data record contains the following information:

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 11-14 illustrates the Tektronix object format.

Figure 11-14. Extended Tektronix Object Format



11.12 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 COFF section load addresses overlap, or a boot table address overlaps another section.

Action This problem may be caused by an incorrect translation from load address to hexadecimal output-file address that is performed by the hex-conversion utility when memory width is less than data width. See Section 11.3 and Section 11.10.

unconfigured memory error

Description The COFF file contains a section whose load address falls outside the memory range defined in the ROMS directive.

Action Correct the ROM range as defined by the ROMS directive to cover the memory range needed, or modify the section load address. Remember that if the ROMS directive is not used, the memory range defaults to the entire processor address space. For this reason, removing the ROMS directive could also be a workaround.

Sharing C/C++ Header Files With Assembly Source

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code.

Topic **Page**

12.1 Overview of the .cdecls Directive

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

See the .cdecls topic for details on the syntax of the .cdecls assembler directive.

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

For example, the following code causes the warning to be issued:

```
.cdecls C,NOLIST
%{
    #define ASMTEST 1
%}

.cdecls C,NOLIST
%{
    #ifndef ASMTEST
        #warn "ASMTEST not defined!"  /* will be issued */
    #endif
%}
```

Therefore, a typical use of the .cdecls block is expected to be a single usage near the beginning of the assembly source file, in which all necessary C/C++ header files are included.

Use the compiler -I*path* (include path) options to specify additional include file paths needed for the header files used in assembly, as you would when compiling C files.

Any C/C++ errors or warnings generated by the code of the .cdecls are emitted as they normally would for the C/C++ source code. C/C++ errors cause the directive to fail, and any resulting converted assembly is not included.

C/C++ constructs that cannot be converted, such as function-like macros or variable definitions, cause a comment to be output to the converted assembly file. For example:

```
; ASM HEADER WARNING - variable definition 'ABCD' ignored
```

The prefix ASM HEADER WARNING appears at the beginning of each message. To see the warnings, either the WARN parameter needs to be specified so the messages are displayed on STDERR, or else the LIST parameter needs to be specified so the warnings appear in the listing file, if any.

Finally, note that the converted assembly code does not appear in the same order as the original C/C++ source code and C/C++ constructs may be simplified to a normalized form during the conversion process, but this should not affect their final usage.

12.2 Notes on C/C++ Conversions

The following sections describe C and ++ conversion elements that you need to be aware of when sharing header files with assembly source.

12.2.1 Comments

Comments are consumed entirely at the C level, and do not appear in the resulting converted assembly file.

12.2.2 Conditional Compilation (#if/#else/#ifdef/etc.)

Conditional compilation is handled entirely at the C level during the conversion step. Define any necessary macros either on the command line (using the compiler -DNAME=value option) or within a .cdecls block using #define. The #if, #ifdef, etc. C/C++ directives are **not** converted to assembly .if, .else, .elseif, and .endif directives.

12.2.3 Pragmas

Pragmas found in the C/C++ source code cause a warning to be generated as they are not converted. They have no other effect on the resulting assembly file. See the .cdecls topic for the WARN and NOWARN parameter discussion for where these warnings are created.

12.2.4 The #error and #warning Directives

These preprocessor directives are handled completely by the compiler during the parsing step of conversion. If one of these directives is encountered, the appropriate error or warning message is emitted. These directives are not converted to .emsg or .wmsg in the assembly output.

12.2.5 Predefined symbol _ _ASM_HEADER_ _

The C/C++ macro _ _ASM_HEADER_ _ is defined in the compiler while processing code within .cdecls. This allows you to make changes in your code, such as not compiling definitions, during the .cdecls processing.

Be Careful With the _ _ASM_HEADER_ _ Macro

Note: You must be very careful not to use this macro to introduce any changes in the code that could result in inconsistencies between the code processed while compiling the C/C++ source and while converting to assembly.

12.2.6 Usage Within C/C++ asm( ) Statements

The .cdecls directive is not allowed within C/C++ asm( ) statements and will cause an error to be generated.

12.2.7 The #include Directive

The C/C++ #include preprocessor directive is handled transparently by the compiler during the conversion step. Such #includes can be nested as deeply as desired as in C/C++ source. The assembly directives .include and .copy are not used or needed within a .cdecls. Use the command line --include_path option to specify additional paths to be searched for included files, as you would for C compilation.

12.2.8 Conversion of #define Macros

Only object-like macros are converted to assembly. Function-like macros have no assembly representation and so cannot be converted. Pre-defined and built-in C/C++ macros are not converted to assembly (i.e., __FILE__, __TIME__, __TI_COMPILER_VERSION__, etc.). For example, this code is converted to assembly because it is an object-like macro:

```
#define NAME Charley
```

This code is not converted to assembly because it is a function-like macro:

```
#define MAX(x,y) (x>y ? x : y)
```

Some macros, while they are converted, have no functional use in the containing assembly file. For example, the following results in the assembly substitution symbol FOREVER being set to the value while(1), although this has no useful use in assembly because while(1) is not legal assembly code.

```
#define FOREVER while(1)
```

Macro values are **not** interpreted as they are converted. For example, the following results in the assembler substitution symbol OFFSET being set to the literal string value 5+12 and **not** the value 17. This happens because the semantics of the C/C++ language require that macros are evaluated in context and not when they are parsed.

```
#define OFFSET  5+12
```

Because macros in C/C++ are evaluated in their usage context, C/C++ printf escape sequences such as \n are not converted to a single character in the converted assembly macro. See Section 12.2.11 for suggestions on how to use C/C++ macro strings.

Macros are converted using the new .define directive (see Section 12.4.2), which functions similarly to the .asg assembler directive. The exception is that .define disallows redefinitions of register symbols and mnemonics to prevent the conversion from corrupting the basic assembly environment. To remove a macro from the assembly scope, .undef can be used following the .cdecls that defines it (see Section 12.4.3).

The macro functionality of # (stringize operator) is only useful within functional macros. Since functional macros are not supported by this process, # is not supported either. The concatenation operator ## is only useful in a functional context, but can be used degenerately to concatenate two strings and so it is supported in that context.

12.2.9 The #undef Directive

Symbols undefined using the #undef directive before the end of the .cdecls are not converted to assembly.

12.2.10 Enumerations

Enumeration members are converted to .enum elements in assembly. For example:

```
enum state { ACTIVE=0x10, SLEEPING=0x01, INTERRUPT=0x100, POWEROFF, LAST};
```

is converted to the following assembly code:

```
state       .enum
ACTIVE      .emember 16
SLEEPING    .emember 1
NTERRUPT    .emember 256
POWEROFF    .emember 257
LAST        .emember 258
            .endenum
```

The members are used via the pseudo-scoping created by the .enum directive:

```
   AC0 = #(state.ACTIVE)
```

The usage is similar to that for accessing structure members, enum_name.member.

This pseudo-scoping is used to prevent enumeration member names from corrupting other symbols within the assembly environment.

12.2.11 C Strings

Because C string escapes such as \n and \t are not converted to hex characters 0x0A and 0x09 until their use in a string constant in a C/C++ program, C macros whose values are strings cannot be represented as expected in assembly substitution symbols. For example:

```
#define MSG "\tHI\n"
```

becomes, in assembly:

```
.define """\tHI\n""",MSG  ; 6 quoted characters! not 5!
```

When used in a C string context, you expect this statement to be converted to 5 characters (tab, H, I, newline, NULL), but the .string assembler directive does not know how to perform the C escape conversions.

You can use the new .cstring directive to cause the escape sequences and NULL termination to be properly handled as they would in C/C++. Using the above symbol MSG with a .cstring directive results in 5 characters of memory being allocated, the same characters as would result if used in a C/C++ strong context. (See Section 12.4.7 for the .cstring directive syntax.)

12.2.12 C/C++ Built-In Functions

The C/C++ built-in functions, such as sizeof( ), are not translated to their assembly counterparts, if any, if they are used in macros. Also, their C expression values are not inserted into the resulting assembly macro because macros are evaluated in context and there is no active context when converting the macros to assembly.

Suitable functions such as $sizeof( ) are available in assembly expressions. However, as the basic types such as int/char/float have no type representation in assembly, there is no way to ask for $sizeof(int), for example, in assembly.

12.2.13 Structures and Unions

C/C++ structures and unions are converted to assembly .struct and .union elements. Padding and ending alignments are added as necessary to make the resulting assembly structure have the same size and member offsets as the C/C++ source. The primary purpose is to allow access to members of C/C++ structures, as well as to facilitate debugging of the assembly code. For nested structures, the assembly .tag feature is used to refer to other structures/unions.

The alignment is also passed from the C/C++ source so that the assembly symbol is marked with the same alignment as the C/C++ symbol. (See Section 12.2.3 for information about pragmas, which may attempt to modify structures.) Because the alignment of structures is stored in the assembly symbol, built-in assembly functions like $sizeof( ) and $alignof( ) can be used on the resulting structure name symbol.

When using unnamed structures (or unions) in typedefs, such as:

```
typedef struct { int a_member; } mystrname;
```

This is really a shorthand way of writing:

```
struct temporary_name { int a_member; };
typedef temporary_name mystrname;
```

The conversion processes the above statements in the same manner: generating a temporary name for the structure and then using .define to output a typedef from the temporary name to the user name. You should use your *mystrname* in assembly the same as you would in C/C++, but do not be confused by the assembly structure definition in the list, which contains the temporary name. You can avoid the temporary name by specifying a name for the structure, as in:

```
typedef struct a_st_name { ... } mystrname;
```

If a shorthand method is used in C to declare a variable with a particular structure, for example:

```
extern struct a_name { int a_member; } a_variable;
```

Then after the structure is converted to assembly, a .tag directive is generated to declare the structure of the external variable, such as:

```
_a_variable .tag a_st_name
```

This allows you to refer to _a_variable.a_member in your assembly code.

12.2.14 Function/Variable Prototypes

Non-static function and variable prototypes (not definitions) will result in a .global directive being generated for each symbol found.

See Section 12.3.1 for C++ name mangling issues.

Function and variable definitions will result in a warning message being generated (see the WARN/NOWARN parameter discussion for where these warnings are created) for each, and they will not be represented in the converted assembly.

The assembly symbol representing the variable declarations will not contain type information about those symbols. Only a .global will be issued for them. Therefore, it is your responsibility to ensure the symbol is used appropriately.

See Section 12.2.13 for information on variables names which are of a structure/union type.

12.2.15 C Constant Suffixes

The C constant suffixes u, l, and f are passed to the assembly unchanged. The assembler will ignore these suffixes if used in assembly expressions.

12.2.16 Basic C/C++ Types

Only complex types (structures and unions) in the C/C++ source code are converted to assembly. Basic types such as int, char, or float are not converted or represented in assembly beyond any existing .int, .char, .float, etc. directives that previously existed in assembly.

Typedefs of basic types are therefore also not represented in the converted assembly.

12.3 Notes on C++ Specific Conversions

The following sections describe C++ specific conversion elements that you need to be aware of when sharing header files with assembly source.

12.3.1 Name Mangling

Symbol names may be mangled in C++ source files. When mangling occurs, the converted assembly will use the mangled names to avoid symbol name clashes. You can use the demangler (dem430) to demangle names and identify the correct symbols to use in assembly.

To defeat name mangling in C++ for symbols where polymorphism (calling a function of the same name with different kinds of arguments) is not required, use the following syntax:

```
extern "C" void somefunc(int arg);
```

The above format is the short method for declaring a single function. To use this method for multiple functions, you can also use the following syntax:

```
extern "C"
  {
      void somefunc(int arg);
      int  anotherfunc(int arg);
      ...
   }
```

12.3.2 Derived Classes

Derived classes are only partially supported when converting to assembly because of issues related to C++ scoping which does not exist in assembly. The greatest difference is that base class members do not automatically become full (top-level) members of the derived class. For example:

```
-------------------------------------------------------
        class base
        {
           public:
                int b1;
        };

        class derived : public base
        {
            public:
               int d1;
        }

-------------------------------------------------------
```

In C++ code, the class derived would contain both integers b1 and d1. In the converted assembly structure "derived", the members of the base class must be accessed using the name of the base class, such as derived.__b_base.b1 rather than the expected derived.b1.

A non-virtual, non-empty base class will have __b_ prepended to its name within the derived class to signify it is a base class name. That is why the example above is derived.__b_base.b1 and not simply derived.base.b1.

12.3.3 Templates

No support exists for templates.

12.3.4 Virtual Functions

No support exists for virtual functions, as they have no assembly representation.

12.4 New Assembler Support

12.4.1 Enumerations (.enum/.emember/.endenum)

New directives have been created to support a pseudo-scoping for enumerations.

The format of these new directives is:

ENUM_NAME	**.enum**
MEMBER1	**.emember** [*value*]
MEMBER2	**.emember** [*value*]
...	
	.endenum

The **.enum** directive begins the enumeration definition and **.endenum** terminates it.

The enumeration name (*ENUM_NAME*) cannot be used to allocate space; its size is reported as zero.

The format to use the value of a member is *ENUM_NAME.MEMBER*, similar to a structure member usage.

The **.emember** directive optionally accepts the value to set the member to, just as in C/C++. If not specified, the member takes a value one more than the previous member. As in C/C++, member names cannot be duplicated, although values can be. Unless specified with .emember, the first enumeration member will be given the value 0 (zero), as in C/C++.

The .endenum directive cannot be used with a label, as structure .endstruct directives can, because the .endenum directive has no value like the .endstruct does (containing the size of the structure).

Conditional compilation directives (.if/.else/.elseif/.endif) are the only other non-enumeration code allowed within the .enum/.endenum sequence.

12.4.2 The .define Directive

The new .define directive functions in the same manner as the existing .asg directive, except that .define disallows creation of a substitution symbol that has the same name as a register symbol or mnemonic. It does not create a new symbol name space in the assembler, rather it uses the existing substitution symbol name space. The syntax for the directive is:

.define *substitution string*, *substitution symbol name*

The .define directive is used to prevent corruption of the assembly environment when converting C/C++ headers.

12.4.3 The .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 12.4.2, which covers the .define directive.

12.4.4 The $defined( ) Directive

The $defined directive returns true/1 or false/0 depending on whether the name exists in the current substitution symbol table or the standard symbol table. In essence $defined returns TRUE if the assembler has any user symbol in scope by that name. This differs from $isdefed in that $isdefed only tests for NON-substitution symbols. The syntax is:

$defined(*substitution symbol name***)**

A statement such as ".if $defined(macroname)" is then similar to the C code "#ifdef macroname".

See Section 12.4.2 and Section 12.4.3 for the use of .define and .undef in assembly.

12.4.5 The $sizeof Built-In Function

The new assembly built-in function $sizeof( ) can be used to query the size of a structure in assembly. It is an alias for the already existing $structsz( ). The syntax is:

$sizeof(*structure name***)**

The $sizeof function can then be used similarly to the C built-in function sizeof( ).

The assembler's $sizeof( ) built-in function cannot be used to ask for the size of basic C/C++ types, such as $sizeof(int), because those basic type names are not represented in assembly. Only complex types are converted from C/C++ to assembly.

Also see Section 12.2.12, which notes that this conversion does not happen automatically if the C/C++ sizeof( ) built-in function is used within a macro.

12.4.6 Structure/Union Alignment & $alignof( )

The assembly .struct and .union directives now take an optional second argument which can be used to specify a minimum alignment to be applied to the symbol name. This is used by the conversion process to pass the specific alignment from C/C++ to assembly.

The assembly built-in function $alignof( ) can be used to report the alignment of these structures. This can be used even on assembly structures, and the function will return the minimum alignment calculated by the assembler.

12.4.7 The .cstring Directive

You can use the new .cstring directive to cause the escape sequences and NULL termination to be properly handled as they would in C/C++.

```
.cstring "String with C escapes.\nWill be NULL terminated.\012"
```

See Section 12.2.11 for more information on the new .cstring directive.

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 -v28 --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 -v28 --symdebug:none --keep_asm input_file
```

The DWARF debugging format consists of the following directives:

- The **.dwtag** and **.dwendtag** directives define a Debug Information Entry (DIE) in the .debug_info section.
- The **.dwattr** directive adds an attribute to an existing DIE.
- The **.dwpsn** directive identifies the source position of a C/C++ statement.
- The **.dwcie** and **.dwendentry** directives define a Common Information Entry (CIE) in the .debug_frame section.
- The **.dwfde** and **.dwendentry** directives define a Frame Description Entry (FDE) in the .debug_frame section.
- The **.dwcfi** directive defines a call frame instruction for a CIE or FDE.

A.2 COFF Debugging Format

COFF symbolic debug is now obsolete. These directives are supported for backwards-compatibility only. The decision to switch to DWARF as the symbolic debug format was made to overcome many limitations of COFF symbolic debug, including the absence of C++ support.

The COFF debugging format consists of the following directives:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the variable or function.
- The **.stag**, **.etag**, and **.utag** directives define structures, enumerations, and unions, respectively. The **.member** directive specifies a member of a structure, enumeration, or union. The **.eos** directive ends a structure, enumeration, or union definition.
- The **.func** and **.endfunc** directives specify the beginning and ending lines of a C/C++ function.
- The **.block** and **.endblock** directives specify the bounds of C/C++ blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source filename.
- The **.line** directive identifies the line number of a C/C++ source statement.

A.3 Debug Directive Syntax

Table A-1 is an alphabetical listing of the symbolic debugging directives. For information on the C/C++ compiler, refer to the *TMS320C28x C/C++ Compiler User's Guide*

Table A-1. Symbolic Debugging Directives

Label	Directive	Arguments
	.block	[*beginning line number*]
	.dwattr	*DIE label*,*DIE attribute name*(*DIE attribute value*)[,*DIE attribute name*(*attribute value*) [, ...]
	.dwcfi	*call frame instruction opcode*[,*operand*[,*operand*]]
CIE label	**.dwcie**	*version* , *return address register*
	.dwendentry	
	.dwendtag	
	.dwfde	*CIE label*
	.dwpsn	" *filename* ", *line number* , *column number*
DIE label	.**dwtag**	*DIE tag name*,*DIE attribute name*(*DIE attribute value*)[,*DIE attribute name*(*attribute value*) [, ...]
	.endblock	[*ending line number*]
	.endfunc	[*ending line number*[,*register mask*[, *frame size*]]]
	.eos	
	.**etag**	*name*[, *size*]
	.file	" *filename* "
	.func	[*beginning line number*]
	.**line**	*line number*[, *address*]
	.member	*name*, *value*[, *type*, *storage class*, *size*, *tag*, *dims*]
	.stag	*name*[, *size*]
	.sym	*name*, *value*[, *type*, *storage class*, *size*, *tag*, *dims*]
	.utag	*name*[, *size*]

XML Link Information File Description

The link step supports the generation of an XML link information file via the --xml_link_info *file* option. This option causes the link step 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 link-step-generated map file.

As the link step evolves, the XML link information file may be extended to include additional information that could be useful for static analysis of link step results.

This appendix enumerates all of the elements that are generated by the link step into the XML link information file.

Topic **Page**

B.1 XML Information File Element Types

These element types will be generated by the link step:

- **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 link step 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 link step (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 COFF Linker       Version x.xx (Jan 6 2004)</banner>
<copyright>Copyright (c) 1996-2004 Texas Instruments Incorporated</copyright>
<link_time>0x43dfd8a4</link_time>
<output_file>hi.out</output_file>
<entry_point>
   <name>_c_int00</name>
   <address>0xaf80</address>
</entry_point>
```

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 link step.

The **<object_component_list>** is a container element enclosing any number of <object_component> elements.

Each **<object_component>** specifies a single object component. Each <object_component> has an id attribute so that it can be referenced directly from other elements, such as a <logical_group>. An <object_component> is a container element enclosing the following elements:

- The **<name>** element names the object component (string).
- The **<load_address>** element specifies the load-time address of the object component (constant).
- The **<run_address>** element specifies the run-time address of the object component (constant).
- The **<size>** element specifies the size of the object component (constant).
- The **<input_file_ref>** element specifies the source file where the object component originated (reference).

Example B-3. Object Component List for the fl-4 Input File

```
<object_component id="oc-20">
   <name>.text</name>
   <load_address>0xac00</load_address>
   <run_address>0xac00</run_address>
   <size>0xc0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
   <name>.data</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
   <name>.bss</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
```

B.2.4 Logical Group List

The **<logical_group_list>** section of the XML link information file is similar to the output section listing in a link-step-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>
```

Glossary

absolute address— An address that is permanently assigned to a TMS320C28x memory location.

absolute lister— A debugging tool that allows you to create assembler listings that contain absolute addresses.

alignment— A process in which the link step 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 link step directive.

allocation— A process in which the link step calculates the final memory addresses of output sections.

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.

assignment statement— A statement that initializes a variable with a value.

autoinitialization— The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at run time— An autoinitialization method used by the link step when linking C code. The link step uses this method when you invoke it with the --rom_model link option. The link step loads the .cinit section of data tables into memory, and variables are initialized at run time.

big endian— An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

binding— A process in which you specify a distinct address for an output section or a symbol.

block— A set of statements that are grouped together within braces and treated as an entity.

.bss section— One of the default object file sections. You use the assembler .bss directive to reserve a specified amount of space in the memory map that you can use later for storing data. The .bss section is uninitialized.

byte— Per ANSI/ISO C, the smallest addressable unit that can hold a character.

C/C++ compiler— A software program that translates C source statements into assembly language source statements.

command file— A file that contains options, filenames, directives, or commands for the link step 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 link step.

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 link step has specified for allocation.

constant— A type whose value cannot change.

cross-reference lister— A utility that produces an output file that lists the symbols that were defined, what file they were defined in, what reference type they are, what line they were defined on, which lines referenced them, and their assembler and link step 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).

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.

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.

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 link step when linking C/C++ code. The link step 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.

link step— A software program that combines object files to form an object module that can be allocated into system memory and executed by the device.

listing file— An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the section program counter (SPC).

little endian— An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

loader— A device that places an executable module into system memory.

macro— A user-defined routine that can be used as an instruction.

macro call— The process of invoking a macro.

macro definition— A block of source statements that define the name and the code that make up a macro.

macro expansion— The process of inserting source statements into your code in place of a macro call.

macro library— An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of .asm.

map file— An output file, created by the link step, that shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.

member— The elements or variables of a structure, union, archive, or enumeration.

memory map— A map of target system memory space that is partitioned into functional blocks.

mnemonic— An instruction name that the assembler translates into machine code.

model statement— Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.

named section— An initialized section that is defined with a .sect directive.

object file— An assembled or linked file that contains machine-language object code.

object library— An archive library made up of individual object files.

object module— A linked, executable object file that can be downloaded and executed on a target system.

operand— An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

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.

partial linking— Linking files in several passes. Incremental linking is useful for large applications because you can partition the application, link the parts separately, and then link all of the parts together.

quiet run— An option that suppresses the normal banner and the progress information.

raw data— Executable code or initialized data in an output section.

relocation— A process in which the link step 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.

structure— A collection of one or more variables grouped together under a single name.

subsection— A relocatable block of code or data that ultimately will occupy continuous space in the memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.

symbol— A string of alphanumeric characters that represents an address or a value.

symbolic debugging— The ability of a software tool to retain symbolic information that can be used by a debugging tool such as a simulator or an emulator.

tag— An optional *type* name that can be assigned to a structure, union, or enumeration.

target memory— Physical memory in a system into which executable object code is loaded.

.text section— One of the default object file sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.

unconfigured memory— Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section— A object file section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss and .usect directives.

UNION— An option of the SECTIONS directive that causes the link step 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.

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 bits-bit addressable location in target memory

Index

B

C

F

G

H

I

M

N

O

S

W

X

Z

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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