TMS320C27x Assembly Language Tools User's Guide

Literature Number: SPRU211D
July 2000

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

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

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Preface

Read This First

About This Manual

The *TMS320C27x Assemby Language Tools User's Guide* tells you how to use these assembly language tools:

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

How to Use This Manual

This book helps you learn to use the Texas Instruments assembly language tools specifically designed for the TMS320C27x devices. You can think of this book as four parts:

- **Introductory information**, consisting of Chapters 1 and 2, gives you an overview of the assembly language development tools. It also discusses common object file format (COFF), which helps you to use the TMS320C27x tools more efficiently. Read Chapter 2, *Introduction to Common Object File Format*, before using the assembler and linker.

- **Assembler description**, consisting of Chapters 3 through 5, contains detailed information about using the assembler. This part of the book explains how to invoke the assembler and discusses source statement format, valid constants and expressions, assembler output, and assembler directives. It also describes the macro language.

- **Additional assembly language tools description**, consisting of Chapters 6 through 10, describes in detail each of the tools provided with the assembler to help you create executable object files. For example, Chapter 7 explains how to invoke the linker, how the linker operates, and how to use linker directives.

- **Reference material**, consisting of Appendixes A through E, provides supplementary information. This part contains technical data about the internal format and structure of COFF object files. It discusses symbolic debugging directives that the TMS320C27x C/C++ compiler uses. Finally, it includes hex-conversion utility examples, assembler and linker error messages, and a glossary.

Notational Conventions

This document uses the following conventions:

- The TMS320C27x core is also referred to as TMS320C27x or 'C27x.

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

  Here is a sample program listing:

```
1 000000                        .data
2 000000 002F                   .byte   47
3 000001 0032                   .byte   50
4 000002 D903                    ADDB AR1 , #3
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered. Syntax that is entered on a command line is centered. Syntax that is used in a text file is left-justified. Here is an example of command-line syntax:

  lnk2000 [*options*] $filename_{1,}$ ...[$filename_{n}$]

  The **lnk2000** command invokes the linker and has two parameters. The first parameter, *options*, is optional (see the next list item for details). The second parameter, $filename_1$, is required, and you can optionally enter more than one filename parameter.

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves. This is an example of a command that has an optional parameter:

  hex2000 [*options*] *filename*

  The **hex2000** command has two parameters. The second parameter, *filename*, is required. The first parameter, *options*, is optional. Since *options* is plural, you can select several options.

- 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 in the left margin of the shaded box, as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, can begin in column 1.

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

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

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte $value_1$ [, ... , $value_n$]

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

  This syntax shows that .byte must have at least one *value* parameter, but you have the option of supplying additional *value* parameters, each separated from the previous one by a comma.

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

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

Related Documentation From Texas Instruments

The following books describe the TMS320C27x and related support tools. To obtain any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, identify the book by its title and literature number (located on the title page):

TMS320C27x DSP CPU and Instruction Set Reference Guide (literature number SPRU220) describes the central processing unit (CPU) and the assembly language instructions of the TMS320C27x 16-bit fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

TMS320C27x Optimizing C/C++ Compiler User's Guide (literature number SPRU212) describes the TMS320C27x C/C++ compiler. This C/C++ compiler accepts ANSI standard C/C++ source code and produces TMS320 assembly language source code for the TMS320C27x device.

TMS320C27xx Translation Assistant User's Guide (literature number SPRU278) describes the TMS320C27xx translation utility and how it fits in with the rest of the TMS320C27xx code development tools. It tells you how to use the translation assistant to translate code you already have for TMS320C2xx devices into code that will run on TMS320C27xx devices.

TMS320C27x Code Composer User's Guide (literature number SPRU296) explains how to use the Code Composer development environment to build and debug embedded real-time DSP applications.

TMS320C27x Code Composer Studio Tutorial (literature number SPRU301) introduces the Code Composer Studio integrated development environment and software tools.

Related Documentation

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

Advanced C: Techniques and Applications, Gerald E. Sobelman and David E. Krekelberg, Que Corporation

American National Standard for Information Systems—Programming Language C X3.159-1989, American National Standards Institute (ANSI standard for C)

Programming in C, Steve G. Kochan, Hayden Book Company

Programming Language C++, ISO/IEC 14882–1998, American National Standards Institute (ANSI standard for C++)

The Annotated C++ Reference Manual, Margaret A. Ellis and Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

The C Programming Language (second edition), Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988, describes ANSI C.

The C++ Programming Language (third edition), Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1997

Understanding and Using COFF, Gintaras R. Gircys, published by O'Reilly and Associates, Inc.

Trademarks

HP-UX is a trademark of Hewlett-Packard Company.

MS-DOS is a registered trademark of Microsoft Corp.

PC-DOS and OS/2 are trademarks of International Business Machines Corp.

Solaris and SunOS are trademarks of Sun Microsystems, Inc.

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

Windows and Windows NT are trademarks of Microsoft Corporation.

XDS is a trademark of Texas Instruments Incorporated.

Contents

Figures

Tables

Examples

Notes

Chapter 1

Introduction to the Software Development Tools

The TMS320C27x™ is supported by a set of software development tools that includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities. This chapter provides an overview of these tools.

The overview shows how tools fit into the general software tools development flow and describes each tool briefly. The TMS320C27x assembly language development tools are:

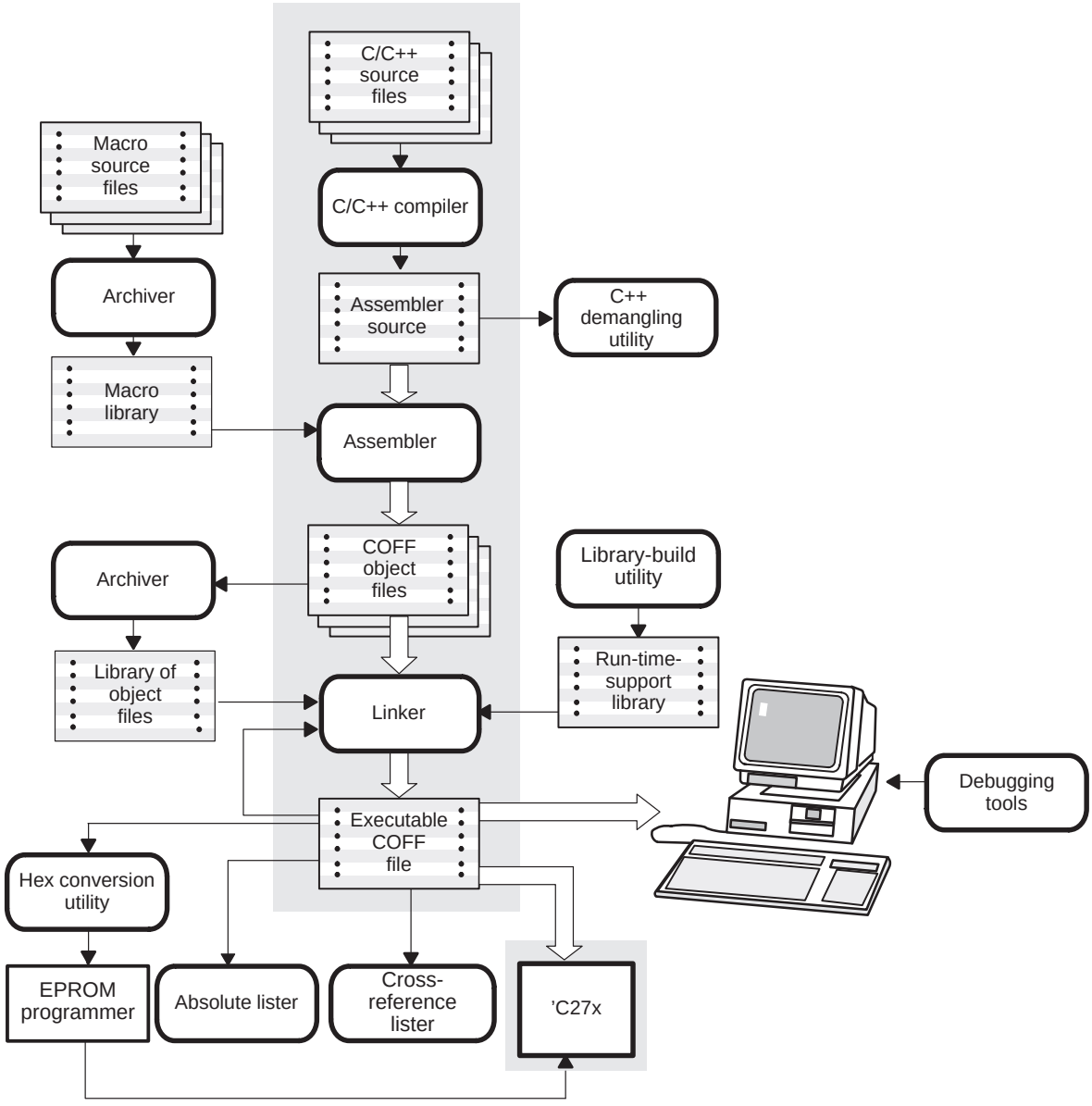
- Assembler
- Archiver
- Linker
- Absolute lister
- Cross-reference lister
- Hex-conversion utility

Topic **Page**

1.1 Software Development Tools Overview

Figure 1–1 shows the TMS320C27x software development flow. The shaded portion highlights the most common development path; the other portions are optional. They are peripheral functions that enhance development.

Figure 1–1. TMS320C27x 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 and C++ source code and produces TMS320C27x 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 programs.
  - The interlist utility interlists C source statements with assembly language output to correlate code produced by the compiler with your source code.

  See the *TMS320C27x Optimizing C/C++ Compiler User's Guide* for more information.

- The **assembler** translates assembly language source files into machine language COFF object files. Source files can contain instructions, assembler directives, and macro directives. You can use assembler directives to control various aspects of the assembly process, such as the source listing format, data alignment, and section content. See Chapter 3, *Assembler Description*, through Chapter 5, *Macro Language*, for more information. See the *TMS320C27x CPU and Instruction Set Reference Guide* for detailed information on the assembly language instruction set.

- The **linker** combines object files into a single executable COFF object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files (created by the assembler) as input. It also accepts archiver library members and output modules created by a previous linker run. Linker directives allow you to combine object file sections, bind sections or symbols to addresses or within memory ranges, and define or redefine global symbols. See Chapter 7, *Linker Description,* for more information.

- The **archiver** allows you to collect a group of files into a single archive file, called a library. For example, you can collect several macros into a macro library. The assembler searches the library and uses the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members. See Chapter 6, *Archiver Description,* for more information.

- You can use the **library-build utility** to build your own customized runtime-support library. See the *TMS320C27x Optimizing C/C++ Compiler User's Guide* for more information.

- The **absolute lister** accepts linked object files as input and creates .abs files as output. You assemble the .abs files to produce a listing that contains absolute addresses rather than relative addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations.

- The **translation assistant** accepts TMS320C2xx™ assembly source input and produces a TMS320C27x assembly source file. Instructions that are not directly translatable are flagged with warning messages to help the programmer complete the translation. See the *TMS320C27x Translation Assistant User's Guide* for more information.

- The **hex-conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer. See Chapter 10, *Hex-Conversion Utility Description,* for more information.

- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definition, and their references in the linked source files. See Chapter 9, *Cross-Reference Lister Description,* for more information.

- The main product of this development process is a module that can be executed in a **TMS320C27x** 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
  - An evaluation module (EVM)

  For information about these debugging tools, see the *TMS320C27x Code Composer Studio User's Guide.*

Chapter 2

Introduction to Common Object File Format

The assembler and linker create object files that can be executed by a TMS320C27x™ device. The format for these object files is called common object file format (COFF).

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

This chapter focuses on the concept and use of sections in assembly language programs. See Appendix A, *Common Object File Format*, for more information on COFF object file structure.

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 ultimately occupies contiguous space in the memory map. Each section of an object file is separate and distinct. COFF object files always contain three default sections:

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

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

There are two basic types of sections:

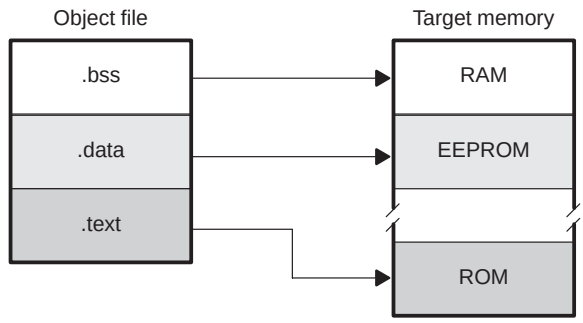
Initialized sections	contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.
Uninitialized sections	reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created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

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

One of the linker's functions is to relocate sections into the target 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 target memory.

Figure 2–1. Partitioning Memory Into Logical Blocks



2.2 How the Assembler Handles Sections

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

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

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

You can create subsections of any section to give you tighter control of the memory map. Subsections are created using the .sect and .usect directives. Subsections are identified with the base section name and a subsection name separated by a colon. See section 2.2.4, *Subsections*, on page 2-7, for more information.

Note: Default Section Directive

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 TMS320C27x memory; they are usually allocated into RAM. These sections have no actual contents in the object file; they simply reserve memory. A program can use this space at run time for creating and storing variables.

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

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

Each time you invoke the .bss or .usect directive, the assembler reserves additional space in the .bss or the named section.

The syntaxes for these directives are:

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

symbol **.usect "***section name* **",** *size in words* [**,** *blocking flag*] [**,** *alignment 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 for which you are reserving space . It can be referenced by any other section and can also be declared as a global symbol (with the .global assembler directive).

size in words is an absolute expression.

- The .bss directive reserves the number of words specified by *size in words* in the .bss section. You must specify a size; there is no default value.
- The .usect directive reserves the number of words specified by *size in words* in *section name*. You must specify a size; there is no default value.

alignment flag is an optional parameter. It specifies the minimum alignment in bytes required by the space allocated.

section name tells the assembler which named section to reserve space in. The *section name* must be enclosed in quotation marks. For more information, see section 2.2.3, *Named Sections*.

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 divert assembly from the current section temporarily. The .bss and .usect directives can appear anywhere in an initialized section without affecting its contents. For an example, see section 2.2.6, *An Example That Uses Sections Directives*, on page 2-8.

The assembler treats uninitialized subsections (created with the .usect directive) in the same manner as uninitialized sections. See section 2.2.4, *Subsections*, on page 2-7 for more information on creating subsections.

2.2.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in TMS320C27x memory when the program is loaded. Each initialized section is independently relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section-relative references.

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

.text

.data

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

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 in memory.

Initialized subsections are created with the .sect directive. The assembler treats initialized subsections in the same manner as initialized sections. See section 2.2.4, *Subsections*, on page 2-7 for more information on creating subsections.

2.2.3 Named Sections

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

For example, repeated use of the .text directive builds up a single .text section in the object file. When linked, this .text section is allocated into memory as a single unit. Suppose there is a portion of executable code (for example, an initialization routine) that you do not want allocated with .text. If you assemble this segment 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*]
.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 .sect and .usect 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 name with .sect.

2.2.4 Subsections

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

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

A subsection is identified by the base section name followed by a colon, then the name of the subsection. For example, you create a subsection called _func within the .text section with the following code:

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

A subsection can be allocated separately or grouped with other sections using the same base name. Using the linker's SECTIONS directive, you can allocate .text:_func separately or with all the .text sections. See section 7.8.1, *SECTIONS Directive Syntax*, on page 7-31 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, *Initialized Sections*, on page 2-5.
- ❏ Uninitialized subsections are created using the .usect directive. See section 2.2.1, *Uninitialized Sections*, on page 2-4.

Subsections are allocated in the same manner as sections. See section 7.8, *The SECTIONS Directive*, on page 7-31 for more information.

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 at that point.

The assembler treats each section as if it began at address 0; the linker relocates each section according to its final location in the memory map. For more information, see section 2.4, *Relocation*, on page 2-14.

2.2.6 An Example That Uses Sections Directives

Example 2–1 shows how you can build COFF sections incrementally, using the sections directives to swap back and forth between sections. You can use sections directives to begin assembling into a section for the first time or to continue assembling into a section that already contains code. In the latter case, the assembler simply appends the new code to the code that is already in the section.

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

Field 1 contains the source code line counter.

Field2 contains the section program counter.

Field 3 contains the object code.

Field 4 contains the original source statement.

See section 3.9, *Source Listings*, on page 3-28 for more information on interpreting the fields in a source listing.

Example 2–1. Using Sections Directives

```
1                ************************************************
2                 **  Assemble an initialized table into .data **
3                ************************************************
4 000000                    .data
5 000000 0011    coeff      .word   011h, 022h, 033h
  000001 0022
  000002 0033
6
7                ************************************************
8                **  Reserve space in .bss for a variable.       **
9                ************************************************
10 000000                   .bss    buffer, 10
11
12                ***********************************************
13                **              Still in .data                **
14                ***********************************************
15 000003 0123    ptr        .word   0123h
16
17                ***********************************************
18                **    Assemble code into the .text section    **
19                ***********************************************
20 000000                   .text
21 000000 D10F    add:       mov     ar1, #0Fh
22 000001 0BA1    aloop:     dec     ar1
23 000002 0009               banz    aloop, ar1--
   000003 FFFF
24
25                ************************************************
26                **  Another initialized table into .data        **
27                ************************************************
28  000004                  .data
29 000004 00AA    ivals      .word   0AAh, 0BBh, 0CCh
   000005 00BB
   000006 00CC
30
31                ************************************************
32                ** Define another section for more variables.   **
33                ************************************************
34 000000         var2       .usect  "newvars", 1
35 000001         inbuf      .usect  "newer", 7
36
37                ************************************************
38                ** Assemble more code into .text.               **
39                ************************************************
40 000004                   .text
41 000004 D10A    end_my:  move      ar1, #0Ah
42 000005 33A1    mloop:     mpy     p,t,ar1
43 000006 28AC               mov     t, #0Ah
   000007 000A
44 000008 3FA1               mov     ar1, p
45 000009 6BFB               sb      end_mpy, OV
```

Field 1 Field 2 Field 3 Field 4

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

.text contains ten 32-bit words of object code.
.data contains five words of object code.
.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 line numbers of the source statements that generated the object code.

Figure 2–2. Object Code Generated by the File in Example 2–1

Line number	Object code	Section
5	0011	.data
5	0022	
5	0033	
15	0123	
29	00AA	
29	00BB	
29	00CC	
21	D10F	.text
22	0BA1	
23	0009	
23	FFFF	
41	D10A	
42	33A1	
43	28AC	
43	000A	
44	3FA1	
45	6BFB	
10	No data 10 words preserved	.bss
34	No data	newvars
35	8 words preserved	

2.3 How the Linker Handles Sections

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

Two linker directives support these functions:

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

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

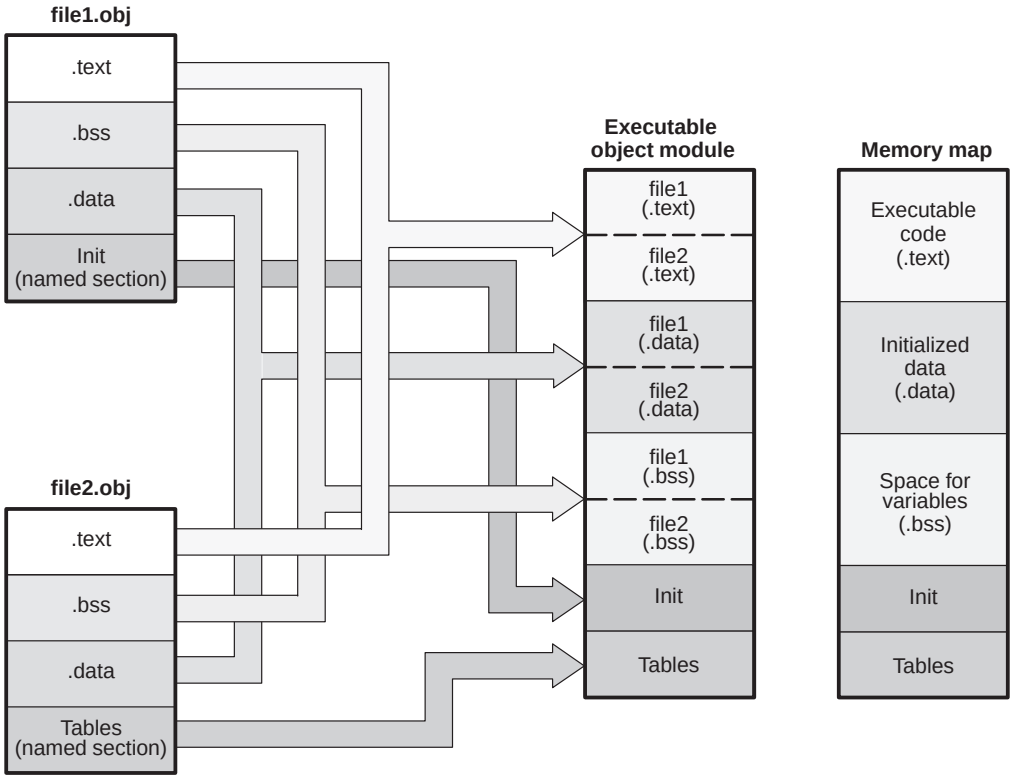
It is not always necessary to use linker directives. If you do not use them, the linker uses the target processor's default allocation algorithm described in section 7.13, *Default Allocation*, on page 7-53. When you *do* use linker directives, you must specify them in a linker command file.

See Chapter 7, *Linker Description,* for more information about linker command files and linker directives.

2.3.1 Default Memory Allocation

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

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



In Figure 2–3, file1.obj and file2.obj have been assembled to be used as linker input. Each contains the .text, .data, and .bss default sections; in addition, each contains a named section. The executable object module shows the combined sections. The linker combines the .text section from file1.obj and file2.obj to form one .text section, then combines the two .data sections and the two .bss sections, and finally places the named sections at the end. The memory map shows how the sections are put into memory.

By default, the linker starts at 0h and allocates sections 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 variables or arrays declared as const. The compiler produces tables of data for autoinitializing global variables; these variables are stored in a named section called .cinit (see page 7-66). For more information on the .const and .cinit sections, see the *TMS320C27x Optimizing C/C++ Compiler User's Guide.*

2.3.2 Placing Sections in the Memory Map

Figure 2–3 illustrates the linker's default method for combining sections. Sometimes you may not want to use the default setup. For example, you may not want all of the .text sections to be combined into a single .text section. Or you may want a named section placed where the .data section would normally be allocated. Most memory maps contain various types of memories (RAM, ROM, EPROM, etc.) in varying amounts; you may want to place a section in a specific type of memory.

For more information on section placement, see section 7.7, *The MEMORY Directive*, on page 7-26 and section 7.8, *The SECTIONS Directive*, on page 7-31.

2.4 Relocation

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

- ❏ Allocating them into the memory map so that they begin at the appropriate address
- ❏ Adjusting symbol values to correspond to the new section addresses
- ❏ Adjusting references to relocated symbols to reflect the adjusted symbol values

The linker uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The linker then uses these entries to patch the references after the symbols are relocated. Example 2–2 contains a TMS320C27x code segment that generates relocation entries.

Example 2–2. An Example of Code That Generates Relocation Entries

```
1                       .global a,b,c,_main
2   000000        a     .usect “seca”,1
3   000000        b     .usect “secb”,1
4   000000        c     .usect “secc”,1
5
6   000000        _main:
7   000000 7700         nop
8   000001        LAB1:                           ;65
9   000001 7700         nop
10  000002        LAB2:                           ;66
11  000002 7700         nop
12  000003        LAB3:                           ;67
13
.
.
.
29  00001d 0900%        addb   acc,#(LAB1 + LAB2)  ;131d
30  00001e 9D00%        addb   ah,#(a + b)         ;66d
31  00001f FE00%        addb   sp,#(a + b)         ;66d
32  000020 0700%        addl   acc,@(a + a + a)    ;9h
33  000021 762F%        and    ifr,#(LAB1 + LAB3)  ;84h
    000022 0000
```

The linker uses the relocation entries to patch the references in the object code:

```
0000005d 0983   ADDB  ACC, #131
0000005e 9d42   ADDB  AH, #66
0000005f fe42   ADDB  SP, #66
00000060 0709   ADDL  ACC, @0x9
00000061  762f  AND   IFR, #0x84
```

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

2.4.1 Run-Time Relocation

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

The linker provides a simple way to handle this. Using the SECTIONS directive, you can optionally direct the linker to allocate a section twice: once 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 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 to the symbol is encountered at run time; this does *not* happen automatically simply because you specify a separate run address. For an example of how to move a block of code at run time, see Example 7–6 on page 7-43.

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

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

For a complete description of run-time relocation, see section 7.9, *Specifying a Section's Run-Time Address*, on page 7-41.

2.5 Loading a Program

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

To run a program, the data in the executable object module must be transferred, or loaded, into target system memory. Several methods can be used for loading a program, depending on the execution environment. Descriptions of two common situations follow:

- The TMS320C27x debugging tools, including the simulator, have built-in loaders. Each of these tools contains a LOAD command that invokes a loader; the loader reads the executable file and copies the program into target memory.
- You can use the hex-conversion utility (hex2000, which is provided as part of the assembly language package) to convert the executable COFF object module into one of several object file formats. You can then use the converted file with an EPROM programmer to burn the program into an EPROM.

2.6 Symbols in a COFF File

A COFF file contains a symbol table that stores information about symbols in the program. The linker uses this table when it performs relocation. Debugging tools can also use the symbol table to provide symbolic debugging.

2.6.1 External Symbols

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

.def is defined in the current module and used in another module.

.ref is referenced in the current module, but defined in another module.

.global may be either of the above.

The following code segment illustrates these definitions.

```
x:  ADD     AR1, #56h    ; Define x
    B       y, UNC       ; Reference y
   .def     x            ; DEF of x
   .ref     y            ; REF of y
```

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

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

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

2.6.2 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 discussed in section 2.6.1). The assembler also creates special symbols that point to the beginning of each section; the linker uses these symbols to relocate references to other symbols.

The assembler does not usually create symbol table entries for other symbols, because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with the .global directive. For symbolic debugging purposes, it is sometimes useful to have entries in the symbol table for each symbol in a program. To accomplish this, invoke the assembler with the –s option (see the –s option description on page 3-5).

Chapter 3

Assembler Description

The TMS320C27x™ assembler translates assembly language source files into machine language object files and performs the following tasks:

- ❏ Processes the source statements in a text file to produce a relocatable object file
- ❏ Produces a source listing (if requested) and provides you with control over this listing
- ❏ Allows you to segment your code into sections and maintain an 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

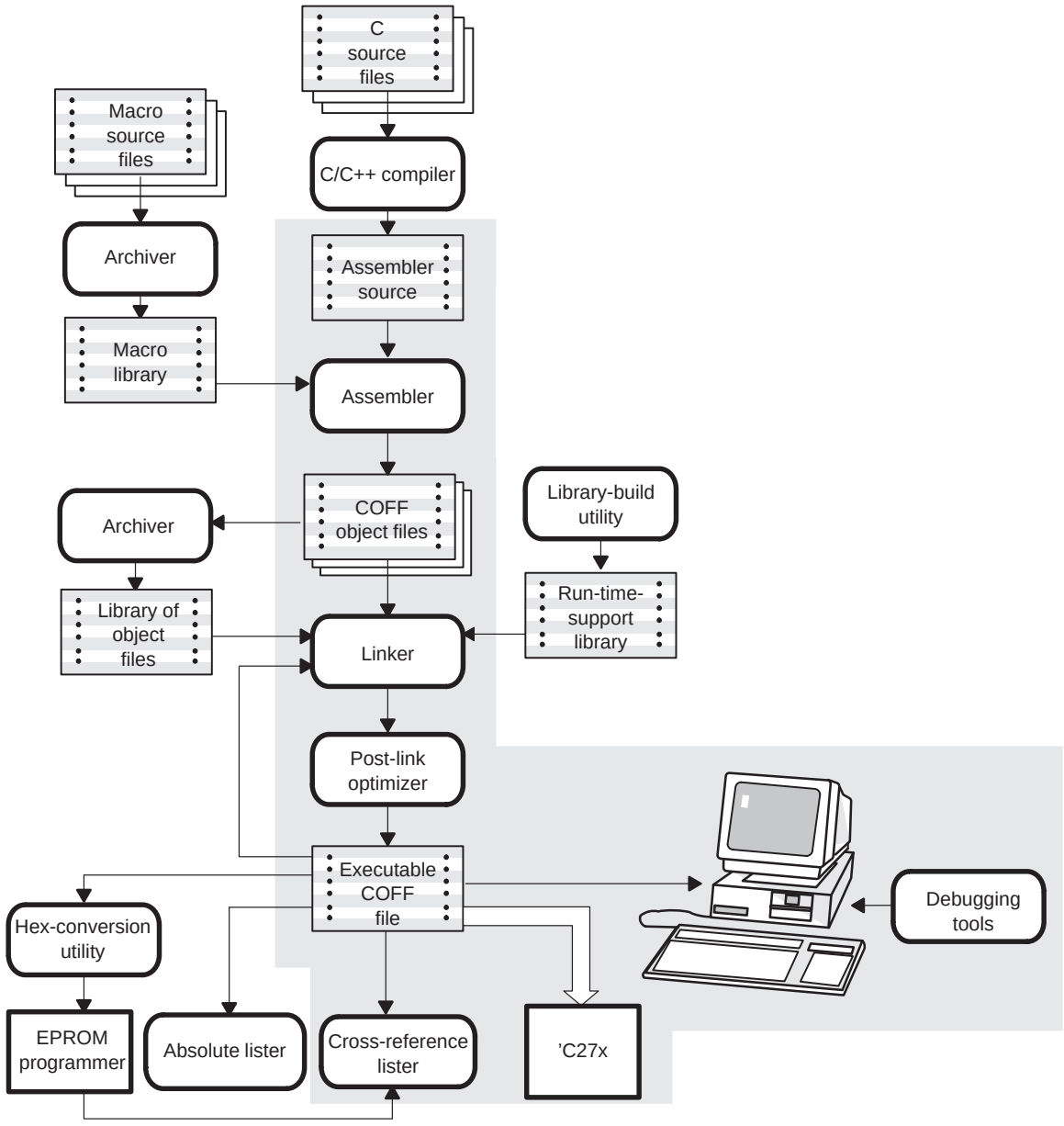
See Chapters 4 and 5 of this book and the *TMS320C27x CPU and Instruction Set User's Guide* for information about source files.

Topic **Page**

3.1 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.

Figure 3–1. The Assembler in the TMS320C27x Software Development Flow



3.2 Invoking the Assembler

Invoke the assembler using the following syntax:

asm2000 [*input file* [*object file* [*listing file*]]] [*options*]

asm2000 is the command that invokes the assembler.

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

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

listing file names the optional listing file that the assembler can create. See section 3.9, *Source Listings*, on page 3-28 for more information on listing files.

- ❏ If you do not supply a *listing file* name, the assembler does *not* create one unless you use the –l (lowercase L) option or –x option.
- ❏ If you supply a *listing file* name but do not supply an extension for that filename, the assembler uses *.lst* as the default extension.

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

–a creates a listing file that provides absolute addresses. When you use –a, the assembler does not produce an object file. Use –a after running the absolute lister.

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

–d **–d***name* [=*value*] sets the *name* symbol. This is equivalent to inserting *name* **.set** [*value*] at the beginning of the assembly file. If *value* is omitted, the symbol is set to 1. For more information, see section 3.7.4, *Defining Symbolic Constants (–d Option)*, on page 3-20.

–f prevents the assembler from adding the default *.asm* file extension if the source file does not have a file extension.

–g adds line number debug information to the object file.

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

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

–i specifies a directory where the assembler can find files named by the .copy, .include, or .mlib directives. The format of the –i option is **–i***pathname*. Each pathname must be preceded by the –i option. For more information, see section 3.3.1, *Using the –i Assembler Option*, on page 3-6.

–l (lowercase L) produces a listing file with the same name as the input file with a *.lst* extension.

–mf allows conditional compilation of 16-bit code with large memory model code. Defines the LARGE_MODEL symbol and sets it to true.

–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.

–mw enables additional assembly-time checking. A warning is generated if a .bss allocation size is greater than 64 words, or a 16-bit immediate operand value resides outside of the –32768 to 65535 range.

–q suppresses the banner and progress information (assembler runs in quiet mode).

–s puts all defined symbols in the object file's symbol table. The assembler usually puts only global symbols into the symbol table. When you use –s, symbols defined as labels or as assembly-time constants are also placed in the table.

–u **–u***name* undefines the predefined constant *name*, which overrides any –d options for the specified constant.

–x 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 –x option, the assembler creates a listing file automatically, naming it with the same name as the input file with a *.lst* extension.

3.3 Naming Alternate Directories for Assembler Input

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

.copy "*filename***"**

.include "*filename***"**

.mlib "*filename***"**

The *filename* names a copy/include file that the assembler reads statements from or a macro library that contains macro definitions. The filename may be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in the following locations in the order given:

1) The directory that contains the current source file. The current source file is the file being assembled when the .copy, .include, or .mlib directive is encountered.
2) Any directories named with the –i assembler option
3) Any directories named with the A_DIR environment variable

Because of this search hierarchy, you can augment the assembler's directory search algorithm by using the –i assembler option (described in section 3.3.1) or the A_DIR environment variable (described in section 3.3.2).

3.3.1 Using the –i Assembler Option

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

asm2000 –i*pathname source filename* [*other options*]

Each –i option names one pathname. In assembly source, you can use the .copy, .include, and .mlib directives without specifying path information. If the assembler does not find the file in the directory that contains the current source file, it searches the paths designated by the –i options.

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

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

Assume the following paths for the copy.asm file:

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

Operating System	Enter
UNIX	**asm2000 –i/320tools/files source.asm**
Windows	**asm2000 –ic:\320tools\files source.asm**

If you invoke the assembler for your system as shown, the assembler first searches for copy.asm in the current directory because source.asm (the input file) is in the current directory. Then the assembler searches in the directory named with the –i option.

3.3.2 Using the A_DIR Environment Variable

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

Operating System	Enter
DOS or OS/2	**set A_DIR=** *pathname*$_1$*;pathname*$_2$*; . . .*
UNIX	**setenv A_DIR "***pathname*$_1$*;pathname*$_2$*; . . .***"**
Windows	**set A_DIR=** *pathname*$_1$*;pathname*$_2$*; . . .*

The *pathnames* are directories that contain .copy/.include files or macro libraries. You can separate the pathnames with a semicolon or with blanks.

3.4 Source Statement Format

TMS320C27x assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments.

Following are examples of source statements:

```
SYM1    .set  2            ; Symbol Two = 2
Begin:  MOV   AR1,#sym1    ; Load AR1 with 2
        .word 016h         ; Initialize a word with 016h
```

The general syntax for source statements is as follows:

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

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. Tab characters are interpreted as blanks.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column *must* begin with a semicolon.

The following sections describe each of the fields.

3.4.1 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 64 alphanumeric characters (A–Z, a–z, 0–9, _, and $). Labels are case sensitive, and the first character cannot be a number. A label can be followed by a colon (:); the colon is not treated as part of the label name. If you 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 subject to the RPT instruction.

When you use a label, its value is the current value of the section program counter (the label points to the statement with which it is associated). For example, if you use the .word directive to initialize several words, a label would point to the first word. In the following example, the label Start has the value 40h.

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

A label on a line by itself is a valid statement. 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 000000           Here:
      2 000000 0003           .word 3
```

3.4.2 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 (||). Pipe symbols indicate instructions that are in parallel with a previous instruction. For example:

```
        RPT
||      Inst2
```

These instructions run in parallel.

The mnemonic field contains one of the following items:

- A machine-instruction mnemonic (such as ADD, MOV, or B)
- An assembler directive (such as .data, .list, or .equ)
- A macro directive (such as .macro, .var, or .mexit)
- A macro call

3.4.3 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 3.7 on page 3-16)
- Constants (see section 3.5 on page 3-11)
- Expressions (a combination of constants and symbols; see section 3.8 on page 3-24)

You must separate operands with commas.

3.4.4 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.

3.5 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.

3.5.1 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 zero-fills the unspecified bits. These are examples of valid binary constants:

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

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

Or, you can use C notation for octal constants:

010	A constant equal to 8_{10} or 8_{16}
0100000	A constant equal to $32\ 768_{10}$ or 8000_{16}
0226	A constant equal to 150_{10} or 96_{16}

3.5.3 Decimal Integers

A decimal integer constant is a string of decimal digits ranging from –2 147 483 648 to 4 294 967 295. These are examples of valid decimal constants:

1000 or 1000d	A constant equal to 1000_{10} or $3E8_{16}$
–32 768	A constant equal to $-32\ 768_{10}$ or 8000_{16}
25	A constant equal to 25_{10} or 19_{16}

3.5.4 Hexadecimal Integers

A hexadecimal integer constant is a string of up to eight 32-bit range 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	A constant equal to 120_{10} or 0078_{16}
0Fh	A constant equal to 15_{10} or $000F_{16}$
37ACh	A constant equal to $14\ 252_{10}$ or $37AC_{16}$

Or, you can use C notation for hexadecimal constants:

0x78	A constant equal to 120_{10} or 0078_{16}
0xF	A constant equal to 15_{10} or $000F_{16}$
0x37AC	A constant equal to $14\ 252_{10}$ or $37AC_{16}$

3.5.5 Character Constants

A character constant is a single character enclosed in *single* quotation marks. The characters are represented internally as 8-bit ASCII characters. Two consecutive single quotation marks are required to represent each single quotation mark that is part of a character constant. A character constant consisting only of two single quotation marks 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 3.6 on page 3-15 discusses character strings.) A character constant represents a single integer value; a string is a sequence of characters.

3.5.6 Assembly-Time Constants

If you use the .set directive (see page 4-65) to assign a value to a symbol, the symbol becomes a constant. To use this constant in expressions, the value that is assigned to it must be absolute. For example:

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

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

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

3.5.7 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

3.6 Character Strings

A character string is a string of characters enclosed in *double* quotation marks. Double quotation marks that are part of character strings are represented by two consecutive double quotation marks. 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

3.7 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 200 alphanumeric characters (A–Z, a–z, 0–9, $, and _). The first character in a symbol cannot be a number, and symbols cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler recognizes ABC, Abc, and abc as three unique symbols. You can override case sensitivity with the –c assembler option (see section 3.2, on page 3-3). A symbol is valid only during the assembly in which it is defined unless you use the .global or .def directive to declare it as an external symbol.

3.7.1 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, as shown in the following example:

```
        .global label1

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

3.7.2 Local Labels

Local labels are special labels whose scope and effect are temporary. A local label can be defined in the following way *name*?, where *name?* is any legal symbol name as previously described. The assembler replaces the question mark with a period followed by a unique number. When the source code is expanded, *you will not see the unique number in the listing file*. Your label appears with the question mark as it did in the source definition. You cannot declare this label as global.

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

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

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

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. However, if you use a local label and a .newblock directive within the macro, the local label is used and reset each time the macro is expanded.

Branches to local labels are not expanded, because local labels are intended to be used only locally and some branches' offsets could be out of range.

Example 3–1 shows code that declares and used local labels legally

Example 3–1. Local Labels

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

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

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

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

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

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

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

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

3.7.3 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 3.7.5, *Predefined Symbolic Constants*, on page 3-22.

3.7.4 Defining Symbolic Constants (–d Option)

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

asm2000 –d*name*=[*value*]

The *name* is the name of the symbol you want to define.

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

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

```
asm2000 –dSYM1=1 –dSYM2=2 –dSYM3=3 –dSYM4=4 value.asm
```

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

Example 3–2. Using Symbolic Constants Defined on Command Line

```
If_4:   .if       SYM4 = SYM2 * SYM2
        .byte     SYM4          ; Equal values
        .else
        .byte     SYM2 * SYM2   ; Unequal values
        .endif

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


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


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

Within assembler source, you can test the symbol defined with the –d option with the following directives:

Type of Test	Directive Usage
Existence	**.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 quotation marks. The quotation marks cause the argument to be interpreted literally rather than as a substitution symbol.

3.7.5 Predefined Symbolic Constants

The assembler has several predefined symbols, including these:

- **$**, the dollar sign character, represents the current value of the section program counter (SPC). $ is a relocatable symbol.
- **The processor symbol** .TMS320C2700, which is always set to 1.
- **CPU control registers**, including:

Register	Description
ACC/AH, AL	Accumulator/accumulator high, accumulator low
AR0	Auxiliary register 0 (index register)
AR1	Auxiliary register 1 (index register)
AR2	Auxiliary register 2
AR3	Auxiliary register 3
AR4	Auxiliary register 4
AR5	Auxiliary register 5
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
ST0	Status register 0
ST1	Status register 1
SP	Stack pointer register
T	Multiplicand register
XAR6, AR6	Extended AR6, auxiliary register 6
XAR7, AR7	Extended AR7, auxiliary register 7

Control registers can be entered as all upper-case or all lower-case characters; that is, IER could also be entered as ier.

3.7.6 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    ”* + AR2 [2]”,  ARG1    ;first arg
    .asg    ”* + AR2 [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
```

For more information about macros, see Chapter 5, *Macro Language*.

3.8 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. Valid expression values are within the 32-bit range –2 147 483 648 to 2 147 483 647 for signed values and the 32-bit range 0 to 4 294 967 295 for unsigned values. Three main factors influence the order of expression evaluation:

1) 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.

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

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

3)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 (see Table 3–1), which is evaluated from right to left.

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

3.8.1 Operators

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

Note: Differences in Precedence From Other TMS320 Assemblers

Some TMS320 processors use a different order of precedence than the TMS320C27x uses. For this reason, different results may be produced from the same source code. The TMS320C27x uses the same order of precedence that the C language uses.

Table 3–1. Order of Precedence of Operators Used in Expressions

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

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

3.8.2 Expression Overflow and Underflow

The assembler checks for overflow and underflow conditions when arithmetic operations *other than* multiplication are performed at assembly time. It issues the warning *Value Truncated* whenever an overflow or underflow occurs.

3.8.3 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:

```
0x1000+X
```

where X was previously defined as an absolute symbol.

3.8.4 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, an absolute value may be compared to an absolute value, but an absolute value may not be compared to a relocatable value.

3.8.5 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 will receive an error message from the linker. For more information on relocation expressions, see section 2.4 on page 2-14.

3.8.5.1 Exceptions to Legal Expressions

- When using the register relative addressing mode, the expression in brackets or parenthesis must be a well-defined expression, as described in section 3.8.3. For example:

```
*+A4[15]
```

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
```

3.9 Source Listings

A source listing shows source statements and the object code they produce. To obtain a listing file, invoke the assembler with the –l (lowercase L) option (see page 3-4).

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. Example 3–3 shows these items 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 illustrated 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. Spacing in this field is determined by the spacing in the source statement.

Example 3–3. Portion of an 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

3.10 Cross-Reference Listings

A cross-reference listing shows symbols and their definitions. To obtain a cross-reference listing, invoke the assembler with the –x option (see page 3-5) or use the .option directive (see page 4-60). The assembler appends the cross-reference to the end of the source listing.

Example 3–4. An Assembler Cross-Reference Listing

```
LABEL                                VALUE       DEFN    REF
.TMS320C27x                       00000001        0
_func                              00000000'      18
var1                               00000000-       4     17
var2                               00000004-       5     18
```

A cross-reference listing contains the following columns:

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 3–2 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 3–2. 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

3.11 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 3–3 lists the instruction to be changed and the change that occurs.

Table 3–3. Smart Encoding for Efficiency

This Instruction...	Is changed to...
`MOV AX/ARX, #8bit`	`MOVB AX/ARX, #8bit`
`ADD AX, #8BitSigned`	`ADDB AX, #8BitSigned`
`CMP AX, #8bit`	`CMPB AX, #8bit`
`ADD ACC, #8bit`	`ADDB ACC, #8bit`
`SUB ACC, #8bit`	`SUBB ACC, #8bit`
`AND AX, #8BitMask`	`ANDB AX, #8BitMask`
`OR AX, #8BitMask`	`ORB AX, #8BitMask`
`XOR AX, #8BitMask`	`XORB AX, #8BitMask`
`B 8BitOffset, cond`	`SB 8BitOffset, cond`
`LB 8BitOffset, cond`	`SB 8BitOffset, cond`
`MOVH loc, ACC << 0`	`MOV loc, AH`
`MOV loc, ACC << 0`	`MOV loc, AL`

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 3–4 shows a list of instructions recognized during intuitive smart encoding and what the instruction is changed to.

Table 3–4. Smart Encoding Intuitively

This instruction...	Is changed to...
MOV XARn, ACC	MOVL XARn, ACC
MOV ACC, XARn	MOVL ACC, XARn
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 XARn	MOV *SP++, XARn
POP XARn	MOV XARn, *––SP
PUSH #16bit	MOV *SP++, #16bit
MPY ACC, T, #8bit	MPYB ACC, T, #8bit

3.12 C-Type Symbolic Debugging for Assembly Variables (–mg option)

When you assembly with the –mg 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 –mg in these ways:

- **Data directives for initialized data**. The assember outputs debugging information for data initialialized 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	INT	SCHAR	UINT
DOUBLE	LDOUBLE	SHORT	ULONG
FLOAT	LONG	UCHAR	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
```

Chapter 4

Assembler Directives

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

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

This chapter is divided into two parts: the first part (sections 4.1 through 4.10) describes the directives according to function, and the second part (section 4.11) is an alphabetical reference.

Topic **Page**

4.1 Directives Summary

In addition to the assembler directives documented here, the TMS320C27x™ software tools support the following directives:

- **Macro directives**, which are discussed in Chapter 5.
- **Symbolic debugging directives**, which are discussed in Appendix B. The C compiler uses these directives for symbolic debugging. Unlike other directives, symbolic debugging directives are not used in most assembly language programs.

> **Note: Labels and Comments in Syntax**
>
> Any source statement that contains a directive can also contain a label and a comment. Labels begin in the first column (they are the only elements, except for comments, that can appear in the first column). 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 summarizes the assembler directives.

Table 4–1. Assembler Directives Summary

(a) Directives that define sections

Mnemonic and Syntax	Description	Page
.bes	Reserves a specified number of bits in the current section	4-66
.bss *symbol*, *size in words* [, *blocking*] [, *alignment*]	Reserves *size* words in the .bss (uninitialized data) section	4-26
.data	Assembles into the .data (initialized data) section	4-34
.sect "*section name***"**	Assembles into a named (initialized) section	4-64
.text	Assembles into the .text (executable code) section	4-73
symbol **.usect "***section name***",** *size in words* [, *blocking*] [,*alignment flag*]	Reserves *size* words in a named (uninitialized) section	4-75
.sblock	Designates section for blocking	4-64

Table 4–1. Assembler Directives Summary (Continued)

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

Mnemonic and Syntax	Description	Page
.byte *value*$_1$ [**,** ... **,** *value*$_n$]	Initializes one or more successive bytes in the current section	4-29
.field *value* [**,** *size*]	Initializes a field of *size* bits (1–32) with *value*	4-40
.float *value*$_1$ [**,** ... **,** *value*$_n$]	Initializes one or more 32-bit IEEE single-precision floating-point constants	4-43
.int *value*$_1$ [**,** ... **,** *value*$_n$]	Initializes one or more 16-bit integers	4-48
.long *value*$_1$ [**,** ... **,** *value*$_n$]	Initializes one or more 32-bit integers	4-53
.pstring	Places 8-bit characters from a character string into the current section	4-68
.space *size*	Reserves *size* bits in the current section; a label points to the beginning of the reserved space	4-66
.string {*expr*$_1$|**"***string*$_1$**"**} [**,** ... **,** {*expr*$_n$|**"***string*$_n$**"**}]	Initializes one or more text strings	4-68
.word *value*$_1$ [**,** ... **,** *value*$_n$]	Initializes one or more 16-bit integers	4-48
.xfloat	Places the floating-point representation of one or more floating-point constants into the current section	4-43
.xlong	Places on or more 32-bit values into consecutive words in the current section	4-53

(c) Directive that aligns the section program counter (SPC)

Mnemonic and Syntax	Description	Page
.align [*size in words*]	Aligns the SPC on a boundary specified by *size in words*, which must be a power of 2; defaults to a 64-word page boundary	4-23

Table 4–1. Assembler Directives Summary (Continued)

(d) Directives that format the output listing

Mnemonic and Syntax	Description	Page
.drlist	Enables listing of all directive lines (default)	4-35
.drnolist	Suppresses listing of certain directive lines	4-35
.fclist	Allows false conditional code block listing (default)	4-39
.fcnolist	Suppresses false conditional code block listing	4-39
.length *page length*	Sets the page length of the source listing	4-50
.list	Restarts the source listing	4-51
.mlist	Allows macro listings and loop blocks (default)	4-57
.mnolist	Suppresses macro listings and loop blocks	4-57
.nolist	Stops the source listing	4-51
.option $option_1$**[,** $option_2$**,** ...]	Selects output listing options; available options are B, L, M, R, T, W, and X	4-60
.page	Ejects a page in the source listing	4-62
.sslist	Allows expanded substitution symbol listing	4-67
.ssnolist	Suppresses expanded substitution symbol listing (default)	4-67
.tab *size*	Sets tab size	4-72
.title "*string***"**	Prints a title in the listing page heading	4-74
.width *page width*	Sets the page width of the source listing to *page width*	4-50

Table 4–1. Assembler Directives Summary (Continued)

(e) Directives that reference other files

Mnemonic and Syntax	Description	Page
.copy ["]*filename*["]	Includes source statements from another file	4-31
.def $symbol_1$ [**,** ... **,** $symbol_n$]	Identifies one or more symbols that are defined in the current module and that can be used in other modules	4-44
.global $symbol_1$ [**,** ... **,** $symbol_n$]	Identifies one or more global (external) symbols	4-44
.include ["]*filename*["]	Includes source statements from another file	4-31
.mlib ["]*filename*["]	Defines macro library	4-55
.ref $symbol_1$ [**,** ... **,** $symbol_n$]	Identifies one or more symbols used in the current module that are defined in another module	4-44

(f) Directives that control conditional assembly

Mnemonic and Syntax	Description	Page
.break [*well-defined expression*]	Ends .loop assembly if *well-defined expression* is true. When using the .loop construct, the .break construct is optional.	4-54
.else	Assembles code block if the .if *well-defined expression* is false. When using the .if construct, the .else construct is optional.	4-46
.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.	4-46
.endif	Ends .if code block	4-46
.endloop	Ends .loop code block	4-54
.if *well-defined expression*	Assembles code block if the *well-defined expression* is true	4-46
.loop [*well-defined expression*]	Begins repeatable assembly of a code block; the loop count is determined by the w*ell-defined expression*.	4-54

Table 4–1. Assembler Directives Summary (Continued)

(g) Directives that define symbols at assembly time

Mnemonic and Syntax	Description	Page
.asg [**"**]*character string*[**"**]**,** *substitution symbol*	Assigns a character string to *substitution symbol*	4-24
.endstruct	Ends structure definition	4-69
.eval *well-defined expression***,** *substitution symbol*	Performs arithmetic on numeric *substitution symbol*	4-24
.label *symbol*	Defines a load-time relocatable label in a section	4-49
symbol **.set** *value*	Equates *value* with *symbol*	4-65
.struct	Begins structure definition	4-69
.tag	Assigns structure attributes to a label	4-69

(h) Miscellaneous directives

Mnemonic and Syntax	Description	Page
.clink [**"***section name***"**]	Enables conditional linking for the current or specified section.	4-30
.emsg *string*	Sends user-defined error messages to the output device; produces no .obj file	4-36
.end	Ends program	4-38
.mmsg *string*	Sends user-defined messages to the output device	4-36
.newblock	Undefines local labels	4-59
.wmsg *string*	Sends user-defined warning messages to the output device	4-36

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

This section explains how the TMS320C27x assembler directives differ from the TMS320C1x/C2x/C2xx/C5x assembler directives.

- The ’C27x .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 ’C27x and the ’C1x/C2x/C2xx/C5x assemblers both align the SPC at the next page boundary. However, the ’C27x .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 ’C27x handles values of 1 to 32 bits, while the ’C1x/C2x/C2xx/C5x assembler handles values of 1 to 16 bits. With the ’C27x 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 ’C27x .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 ’C27x initializes one character per word; the ’C1x/C2x/C2xx/C5x assembler directive .string, packs two characters per word. The ’C27x .pstring directive packs two characters per word.
- The following directives are new with the ’C27x assembler and are not supported by the ’C1x/C2x/C2xx/C5x assembler:

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

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, *Introduction to Common Object File Format*, discusses COFF sections in detail.

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

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

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

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

Example 4–1. Sections Directives

```
1                ************************************************
2                *     Start assembling into the .text section     *
3                ************************************************
4  000000                .text
5  000000 0001           .word   1, 2
   000001 0002
6  000002 0003           .word   3, 4
   000003 0004
7
8                ************************************************
9                *     Start assembling into the .data section     *
10                ************************************************
11 000000                .data
12 000000 0009           .word   9, 10
   000001 000A
13 000002 000B           .word   11, 12
   000003 000C
14
15                ************************************************
16                *     Start assembling into a named,              *
17                *     initialized section, var_defs               *
18                ************************************************
19 000000                .sect   ”var_defs”
20 000000 0011           .word   17, 18
   000001 0012
21
22                ************************************************
23                *     Resume assembling into the .data section    *
24                ************************************************
25 000004                .data
26 000004 000D           .word   13, 14
   000005 000E
27 000000                .bss    sym, 19    ; Reserve space in .bss
28 000006 000F           .word   15, 16     ; Still in .data
   000007 0010
29
30               ************************************************
31               *     Resume assembling into the .text section    *
32               ************************************************
33 000004                .text
34 000004 0005           .word   5, 6
   000005 0006
35 000000       usym     .usect  ”xy”, 20   ;  Reserve space in xy
36 000006 0007           .word   7, 8       ; Still in .text
   000007 0008
```

4.4 Directives That Initialize Constants

Several directives assemble values for the current section:

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

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

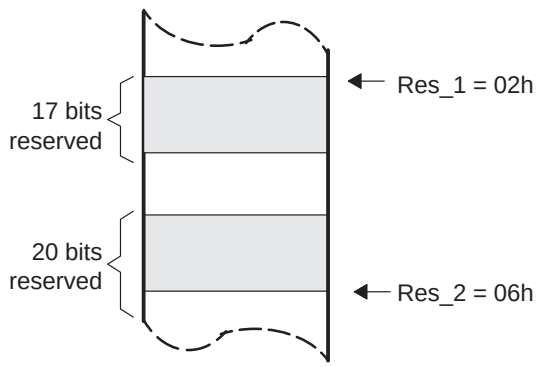
  - When you use a label with .space, it points to the *first* word that contains reserved bits.
  - When you use a label with .bes, it points to the *last* word that contains reserved bits.

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

```
1                **  .space and .bes directives
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 4–1. The .space and .bes Directives

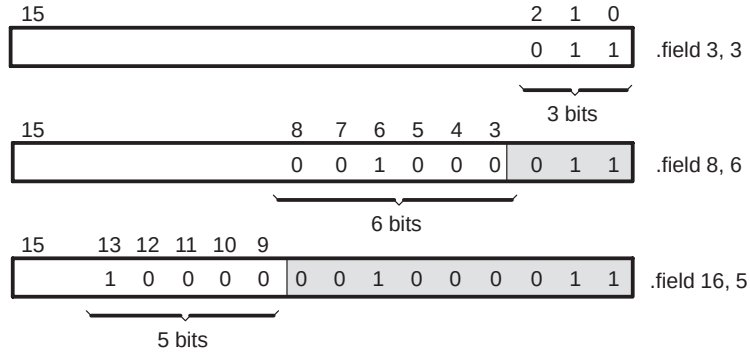


- **.byte** places one or more 16-bit values into consecutive words of the current section. This directive is similar to .word, except that the width of each value is restricted to eight bits.
- The **.field** directive places a single value into a specified number of bits in the current word. With .field, you can pack multiple fields into a single word; the assembler does not increment the SPC until a word is filled.

Figure 4–2 shows how fields are packed into a word. For this example, assume the following code has been assembled; 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 4–2. 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 stores it in a word in the current section.
- The **.int**, and **.word** directives place one or more 16-bit values into consecutive 16-bit fields in the current section.
- The **.long** and **.xlong** directives place one or more 32-bit values into consecutive 32-bit fields in the current section.
- The **.string** and **.xstring** directives place 8-bit characters from one or more character strings into the current section one character per word. This directive is similar to .byte.

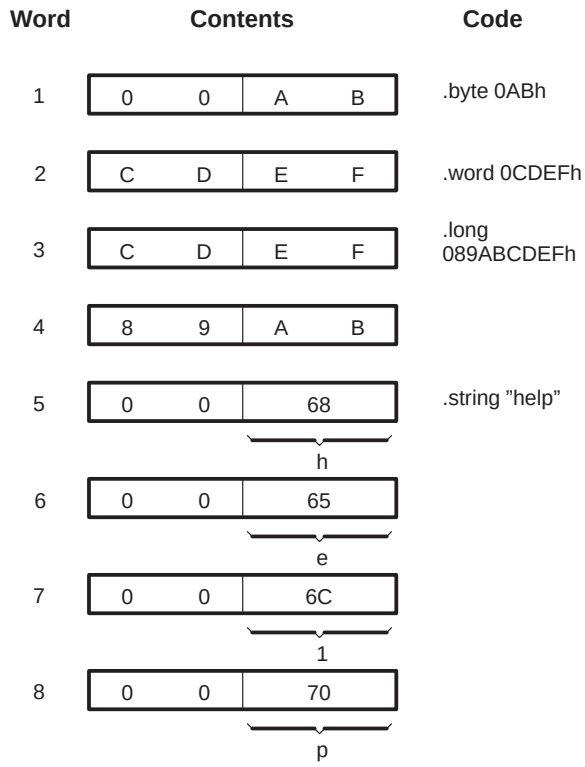
Note: Directives That Initialize Constants When Used in a .struct/.endstruct Sequence

The .byte, .int, .long, .word, .string, .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 about the .struct/.endstruct directives, see page 4-69.

Figure 4–3 compares the .byte, .word, and .string directives. For this example, assume the following code has been assembled:

```
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 4–3. Initialization Directives



4.5 Directive That Aligns the Section Program Counter

The **.align** directive aligns the SPC to the specified word boundary. This ensures that the code following the directive begins on that boundary. If the SPC is already aligned at the selected boundary, it is not incremented. Operands for the .align directive must equal a power of 2 between 1 and 65 536. For example:

Operand of 1 aligns SPC to word boundary

2 aligns SPC to long word/even boundary

64 aligns SPC to page boundary

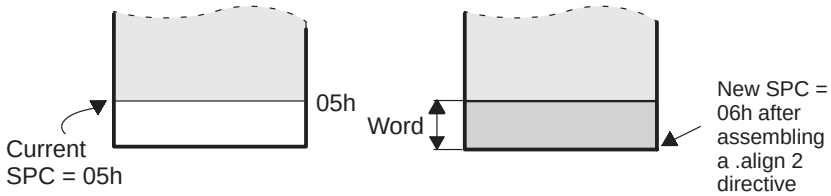
The .align directive with no operands defaults to 64, that is, to a page boundary.

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

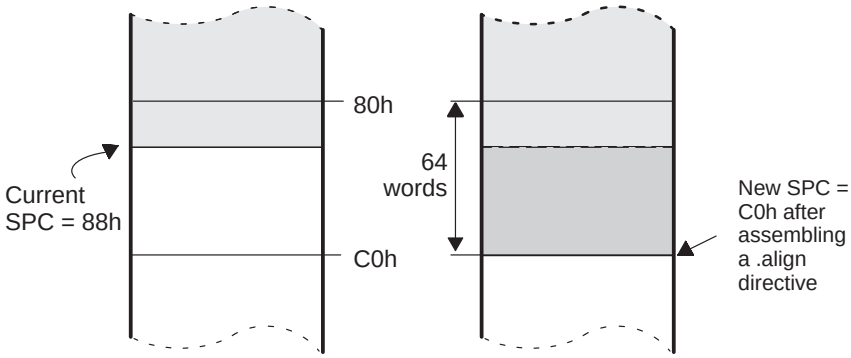
```
1 000000 0002          .field  2,3
2 000000 005A          .field  11,8
3                      .align  2
4 000002 0065          .string "errorcnt"
  000003 0072
  000004 0072
  000005 006F
  000006 0072
  000007 0063
  000008 006E
  000009 0074
5                      .align
6 000040 0004         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

Figure 4–4. The .align Directive

(a) Result of .align 2



(b) Result of .align without an argument



4.6 Directives That Format the Output Listing

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.

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

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

- The source code listing includes false conditional blocks that do not generate code. The **.fclist** and **.fcnolist** directives turn this listing on and off. You can use the .fclist directive to list false conditional blocks exactly as they appear in the source code. You can use the .fcnolist directive to list only the conditional blocks that are actually assembled.

- The **.length** directive controls the page length of the listing file. You can use this directive to adjust listings for various output devices.

- The **.list** and **.nolist** directives turn the output listing on and off. You can use the .nolist directive to prevent the assembler from printing selected source statements in the listing file. Use the .list directive to turn the listing on again.

- The source code listing includes macro expansions and loop blocks. The **.mlist** and **.mnolist** directives turn this listing on and off. You can use the .mlist directive to print all macro expansions and loop blocks to the listing and the .mnolist directive to suppress this listing.

- The **.option** directive controls certain features in the listing file. This directive has the following operands:

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

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

  M turns off macro expansions in the listing.

  R resets the B, H, L, M, T, and W directives (turns off the limits of B, H, 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 –x option (see page 3-5).

- 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:

- The **.copy** and **.include** directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the .copy/.include file, it resumes reading source statements from the current file. The statements read from a copied file are printed in the listing file; the statements read from an included file are *not* printed in the listing file.

- The **.def** directive identifies a symbol that is defined in the current module and that can be used in another module. The assembler includes the symbol in the symbol table.

- The **.global** directive declares a symbol external so that it is available to other modules at link time. (For more information about global symbols, see section 2.6.1, *External Symbols*, on page 2-18.) The .global directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. The linker resolves an undefined global symbol reference only if the symbol is used in the program.

- The **.mlib** directive supplies the assembler with the name of an archive library that contains macro definitions. When the assembler encounters a macro that is not defined in the current module, it searches for it in the macro library specified with .mlib.

- The **.ref** directive identifies a symbol that is used in the current module but defined in another module. The assembler marks the symbol as an undefined external symbol and enters it in the object symbol table so the linker can resolve its definition. The .ref directive forces the linker to resolve a symbol reference.

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.
.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 *well-defined expression* evaluates to the loop count.
[**.break** [*well-defined expression*]]	tells the assembler to continue to repeatedly assemble 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 3.8.4, *Conditional Expressions*, on page 3-26.

4.9 Directives That Define Symbols at Assembly Time

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

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

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

.byte coefficients
```

- 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, as in the following example:

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

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

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

```
bval   .set   1000h
       .long  bval, bval*2, bval+12
       MOV    AL, #bval
```

  The .set directive produces no object code.

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

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

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

```
COORDT .struct
X       .int
Y       .int
T_LEN   .endstruct

COORD   .tag    COORDT
        ADD     ACC, @COORD.Y

        .bss    COORD, T_LEN
```

4.10 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.clink** directive sets the STYP_CLINK flag in the type field for the named section. The .clink directive can be applied to initialized or uninitialized sections. The STYP_CLINK flag enables conditional linking by telling the linker to leave the section out of the final COFF output of the linker if there are no references found to any symbol in the section.
- The **.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 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. (For more information, see section 3.7.2, *Local Labels*, on page 3-16.)

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 information about using the error and warning directives in macros, see section 5.7, *Producing Messages in Macros*, on page 5-17.

4.11 Directives Reference

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

Syntax

.align [*size in words*]

Description

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size in words* parameter. The *size* may be any power of 2, although only certain values are useful for alignment. An operand of 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 NOP up to the next x-word boundary.

Operand of 1 aligns SPC to word boundary

2 aligns SPC to long word/even boundary

64 aligns SPC to page boundary

Using the .align directive has two effects:

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

Example

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

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

Syntax

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

Description

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

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

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

- The *well-defined expression* is an alphanumeric expression consisting of legal values that have been previously defined, so that the result is an absolute.

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

```
       1              .sslist      ; show expanded
       2                           ; substitution symbols
       3               .asg    XAR6, FP
       4
       5 000000 0964    ADD     ACC, #100
       6 000001 778     NOP     *FP++
#                       NOP     *XAR6++
       7
       8                .asg      0, x
       9                .loop     5
      10                .eval     x+1, x
      11                .word     x
      12                .endloop
1                       .eval     x+1, x
#                       .eval     0+1, x
1         000002 0001   .word     x
#                       .word     1
1                       .eval     x+1, x
#                       .eval     1+1, x
1         000003 0002   .word     x
#                       .word     2
1                       .eval     x+1, x
#                       .eval     2+1, x
1         000004 0003   .word     x
#                       .word     3
1                       .eval     x+1, x
#                       .eval     3+1, x
1         000005 0004   .word     x
#                       .word     4
1                       .eval     x+1, x
#                       .eval     4+1, x
1         000006 0005   .word     x
#                       .word     5

 No Errors, No Warnings
```

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 variables in RAM.

- The *symbol* is a required parameter. It defines a label that points to the first location reserved by the directive. The symbol name should correspond to the variable for which you are reserving space.
- The *size in words* is a required parameter; it must be an absolute expression. The assembler allocates *size in 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. This flag 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 3.12, *C-Type Symbolic Debugging for Assembly Variables (–mg option*), on page 3-34 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.

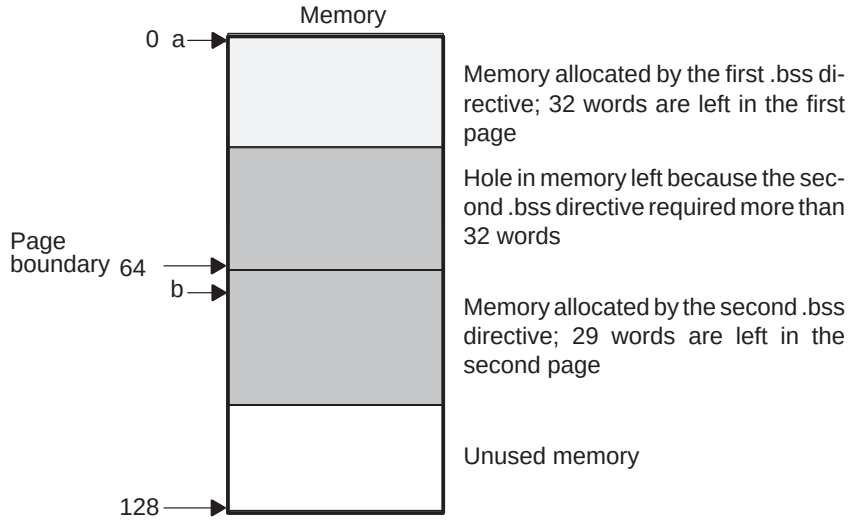
- If you do not request blocking, the memory is allocated at the current SPC.
- If you request blocking, the assembler checks to see whether there is enough space between the current SPC and the page boundary. If there is not enough space, the assembler creates another hole and allocates the space on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



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

Example

In this example, the .bss directive is used to allocate space for two variables, TEMP and ARRAY. The symbol TEMP points to 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

No Errors, No Warnings
```

Syntax

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

Description

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

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

Values are not packed or sign-extended; each byte occupies the 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 in which the assembler places the first byte.

When you use .byte in a .struct/.endstruct sequence, .byte defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, see section 4.9, *Directives That Define Symbols at Assembly TIme*, on page 4-19.

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
```

Syntax

.clink [”*section name*”]

Description

The **.clink** directive sets up conditional linking for a section by setting the STYP_CLINK flag in the type field for *section name*. The .clink directive can be applied to initialized or uninitialized sections.

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 quotation marks. A *section name* can contain a section name in the form *section name*:*subsection name*.

The STYP_CLINK flag tells the linker to leave the section out of the final COFF output of the linker if no references are found to any symbol in the section.

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

Example

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

```
 1 000000                .sect   ”Vars”
 2                ; Vars section is conditionally linked
 3                       .clink
 4
 5 000000 001A  X:       .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 E000+         MOV     ACC, @X
21 000002 3100          MOV     P, #0
22 000003 FF59          CMP     ACC, P
```

Syntax

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

Description

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

1) Stops assembling statements in the current source file

2) Assembles the statements in the copied/included file

3) Resumes assembling statements in the main source file, starting with the statement that follows the .copy or .include directive

The *filename* is a required parameter that names a source file. It may be enclosed in double quotation marks and must follow operating system conventions. You can specify a full pathname (for example, c:\dsp\file1.asm). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file
2) Any directories named with the –i assembler option
3) Any directories specified by the environment variable A_DIR

For more information about the –i option and A_DIR, see section 3.3, *Naming Alternate Directories for Assembler Input*, on page 3-6.

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

Example 1

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

The original file, copy.asm, contains a .copy statement copying the file byte.asm. When copy.asm assembles, the assembler copies byte.asm into its place in the listing. The copied 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 the assembly of byte.asm, the assembler returns to copy.asm to assemble its remaining statement.

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

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

Listing file:

```
1 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                 .pstring "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 included file)	word2.asm (second included 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
```

Syntax

```
.data
```

Description

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

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

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

Example

In this example, code is assembled into the .data and .text sections.

```
1                ****************************************
2                **     Reserve space in .data.        **
3                ****************************************
4  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
```

Syntax

```
.drlist
.drnolist
```

Description

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

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

The **.drnolist** directive suppresses the printing of the following directives to the listing file:

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

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

Example

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

Source file:

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

.drnolist

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

Listing file:

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

Syntax

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

Description

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

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

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

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

Example

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

Source file:

```
          .global     PARAM
MSG_EX    .macro parm1
          .if    $symlen(parm1) = 0
          .emsg  ”ERROR –– MISSING PARAMETER”
          .else
           add   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
```

Syntax

```
.end
```

Description

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

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

Note: Ending a Macro

Use .endm to end a macro.

Example

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

Source File:

```
START:   .space  300
TEMP     .set    15
         .bss    LOC1, 48h
         ABS     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

 No Errors, No Warnings
```

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 listing of conditional blocks:

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
```

Syntax

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

Description

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

- The *value* is a required parameter; it is an expression that is evaluated and placed in the field. If the value is relocatable, *sizein bits* must be 22 or greater.
- The *sizein 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 in bits*, the assembler assumes that 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 into *size in bits*, the assembler truncates the value and issues an error message. For example, .field 3,1 causes the assembler to truncate the value 3 to 1; the assembler also prints the message:

```
***warning – value truncated.
```

Successive .field directives pack values into the specified number of bits starting at the current word. Fields are packed starting at the 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.

When you specify a size greater than 16 bits, the assembler fills the word at the lower address, then begins filling the least significant bits of the word at the higher address.

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

When you use .field in a .struct/.endstruct sequence, .field defines a member's size; it does not initialize memory. For more information about .struct/ .endstruct, see page 4-69.

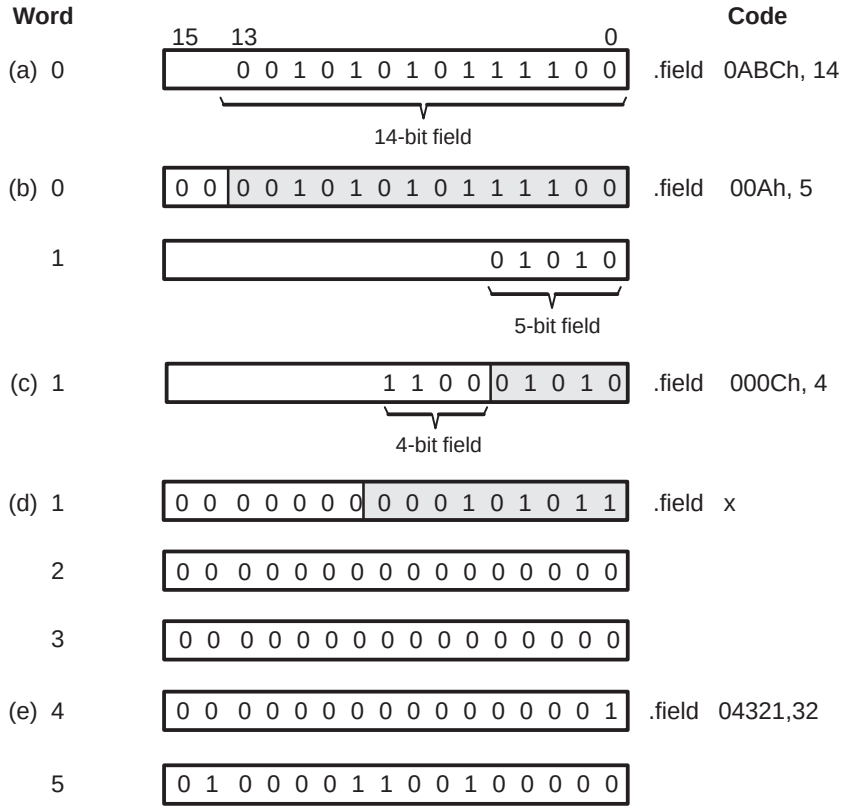
Example

This example shows how fields are packed into a word. The SPC does not change until a word is filled and the packing of the next word begins. For another example of the .field directive, see page 4-11.

```
 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 shows how the directives in this example affect memory.

Figure 4–6. The .field Directive



Syntax

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

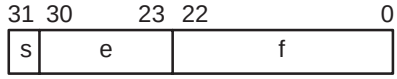
Description

The **.float** and **.xfloat** directives place the floating-point representation of one or more floating-point constants into the current section. The *value* must be a floating-point constant or a symbol that has been equated to a floating-point constant. Each constant is converted to a floating-point value in IEEE single-precision 32-bit format. Floating point constants are aligned on the long-word boundaries unless the .xfloat directive is used. The .xfloat directive performs the same function as the .float directive but does not align the result on the long-word boundary.

The 32-bit value consists of three fields:

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

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



When you use .float in a .struct/.endstruct sequence, .float defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, see section 4.9, *Directives That Define Symbols at Assembly Time*, on page 4-19.

Example

This example shows the .float directive.

```
1 000000 5951          .float  –1.0e25
  000001 E904
2
3 000002 0000          .float  3
  000003 4040
4
5 000004 0000          .float  123
  000005 42F6
6
```

Syntax

.global $symbol_1$ [, ... , $symbol_n$]
.def $symbol_1$ [, ... , $symbol_n$]
.ref $symbol_1$ [, ... , $symbol_n$]

Description

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

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

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

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

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

A symbol may be declared global for two reasons:

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

Example

This example shows four files:

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

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

file1.lst:

```
 1              ; Global symbol defined in this file
 2                     .global INIT
 3              ; Global symbols defined in file2.lst
 4                     .global X, Y, Z
 5 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
```

Syntax

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

Description

The following directives provide conditional assembly:

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

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

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

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

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

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

For information about relational operators, see section 3.8.4, *Conditional Expressions*, on page 3-26.

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
```

Syntax

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

Description

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

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

You can use as many values as fit on a single line. If you use a label, it points to the first word that is initialized.

When you use .int or .word in a .struct/.endstruct sequence, .int or .word defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, see section 4.9, *Directives That Define Symbols at Assembly Time*, on page 4-19.

Example 1

In this example, the .int directive is used 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
```

Syntax

.label *symbol*

Description

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

For some applications, it is desirable to have a section load at one address and run at a different address. For example, you may wish to load a block of performance-critical code into slower off-chip memory to save space and then move the code to high-speed on-chip memory to run it.

Such a section is assigned two addresses at link time: a load address and a run address. All labels defined in the section are relocated to refer to the runtime address so that references to the section (such as branches) are correct when the code runs.

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

Example

This example shows the use of a loadtime address label.

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

For more information about assigning runtime and loadtime addresses in the linker, see section 7.9, *Specifying a Section's Runtime Address*, on page 7-41.

Syntax

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

Description

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

- Default length: 60 lines
- Minimum length: 1 line
- Maximum length: 32 767 lines

The **.width** directive sets the page width of the output listing file. It affects the next line assembled and the lines that follow; you can reset the page width with another .width directive.

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

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

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

Example

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

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

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

Syntax

```
.list
.nolist
```

Description

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

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

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

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

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

Note: Creating a Listing File (–l Option)

If you do not request a listing file when you invoke the assembler, the assembler ignores the .list directive.

Example

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

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 (copy file)
      .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
```

Syntax

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

Description

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

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

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

When you use .long in a .struct/.endstruct sequence, .long defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, see section 4.9, *Directives That Define Symbols at Assembly Time*, on page 4-19.

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:
```

Syntax

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

Description

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

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

The **.break** directive and it's well-defined *expression* are optional. When the expression is false (0), the loop continues. When the expression is true (nonzero) or omitted, the assembler breaks the loop and assembles the code after the .endloop directive.

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

```
       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
```

Syntax

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

Description

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

The *filename* of a macro library member must be the same as the macro name, and its extension must be .asm. The filename must follow host operating system conventions; it may be enclosed in double quotes. You can specify a full pathname (for example, c:\dsp\macs.lib). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file
2) Any directories named with the –i assembler option
3) Any directories specified by the environment variable A_DIR

For more information about the –i option and the environment variable, see section 3.3, *Naming Alternate Directories for Assembler Input*, on page 3-6.

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

Example

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

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

```
       1                 .mlib    "mac.lib"
       2
       3        * Macro call
       4 000000         inc1    AL
1        000000 9C01    ADD     AL,#1
       5
       6        * Macro call
       7 000001         dec1    AR1
1        000001 08A1    SUB     AR1,#1
         000002 FFFF

No Errors, No Warnings
```

Syntax

```
.mlist
.mnolist
```

Description

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

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

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

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

Example

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

```
      1              STR_3  .macro   P1, P2, P3
      2                     .string ":p1:", ":p2:", ":p3:"
      3                     .endm
      4
      5 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
```

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.

After a local label has been defined and (perhaps) used, you should use the .newblock directive to reset it. The .text, .data, and named sections also reset local labels. Local labels that are defined within an include file are not valid outside of the local file. For a description of local labels, see section 3.7.2 on page 3-16.

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 000000 F800!         MOV    DP, #ADDRA
 5
 6 000001 E000! LABEL1: MOV    ACC, @ADDRA
 7 000002 1976          SUB    ACC, #B
 8 000003 6403          B      $1, LT
 9 000004 9600!         MOV    @ADDRB, ACC
10 000005 6F02          B      $2
11
12 000006 E000! sym1?   MOV    ACC, @ADDRA
13 000007 A000! sym2?   ADD    ACC, @ADDRC
14                      .newblock      ; Undefine $1 to
                                           use again.
15 000008 6402          B      $1, LT
16 000009 9600!         MOV    @ADDRC, ACC
17 00000a 7700  $1      NOP

No Errors, No Warnings
```

Syntax

```
.option  option list
```

Description

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

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

Options are not case sensitive.

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

```
 1                    ****************************************
 2                    ** Limit the listing of .byte, .word, **
 3                    ** .long, and .string directives to 1 **
 4                    **         to 1 line each.            **
 5                    ****************************************
 6                         .option B, W, L, T
 7 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
```

Syntax

```
.page
```

Description

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

Example

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

Source file:

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

Listing file:

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

       2            ;        .
       3            ;        .
       4            ;        .
TMS320C27x COFF Assembler       Version x.xx     Day      Time      Year
 Copyright (c) xxxx–xxxx    Texas Instruments Incorporated

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

Syntax

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

Description

The **.sblock** directive designates sections for blocking. Blocking is an address alignment mechanism similar to page alignment, but weaker. A blocked section 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
```

Syntax

.sect "*section name*"

Description

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

The *section name* identifies a section that the assembler assembles code into. The name can be up to 200 characters and must be enclosed in double quotation marks. A section name can contain a subsection name in the form *section name*:*subsection name*.

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

Example

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

```
 1                **   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

No Errors, No Warnings
```

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.

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   .equ    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

   No Errors, No Warnings
```

Syntax

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

Description

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

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

Example

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

```
1                ******************************************
2                **  Begin assembling into .text section.   **
3                ******************************************
4   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_
```

Syntax

```
.sslist
.ssnolist
```

Description

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

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

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

By default, all substitution symbol expansion in the listing file is inhibited. Lines with the pound (#) character denote expanded substitution symbols.

Example

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

```
       1 000000               .bss    ADDRX, 1
       2 000001               .bss    ADDRY, 1
       3 000002               .bss    ADDRA, 1
       4 000003               .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 000000 E084          ADD2     ADD2, ADDRX, ADDRY
      13
      14                      .sslist
      15 000001 9684          MOV     *AR4++, ACC
      16 000002 9680          ADD2     ADDRA, ADDRB
      17

No Errors, No Warnings
```

Syntax

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

Description

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

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

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

The assembler truncates any values that are greater than eight bits. You may have up to 100 operands, but they must fit on a single source statement line.

If you use a label, it points to the location of the first 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 about .struct/.endstruct, see section 4.9, *Directives That Define Symbols at Assembly Time*, on page 4-19.

Example

This example shows 8-bit values 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
```

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 enables you to group similar data elements together and then let the assembler calculate the element offset. This is similar to a C structure or a Pascal record.

Note: .struct Does Not Allocate Memory

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

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

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

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

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

mem_n is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.

element is one of the following descriptors: .string, .byte, .word, .float, .tag, or .field. All of these except .tag are typical directives that initialize memory. Following a .struct directive, these directives describe the structure element's size. They do not allocate memory. A .tag directive is a special case because a *stag* must be used (as in the definition).

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

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

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

The only directives that can appear in a .struct/.endstruct sequence are element descriptors, conditional assembly directives, and the .align directive, which aligns the member offsets on word boundaries. Empty structures are illegal.

These examples show various uses of the .struct, .tag, and .endstruct directives.

Example 1

```
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
```

Syntax

```
.tab  size
```

Description

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

Example

Each of the following lines consists of a single tab character followed by an NOP instruction.

Source file:

```
; default tab size
   NOP
   NOP
   NOP

     .tab 4
   NOP
   NOP
   NOP

     .tab 16
   NOP
   NOP
   NOP
```

Listing file:

```
      1                          ; default tab size
      2 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
```

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 specify another section with the .data or .sect directives. For more information about COFF sections, see Chapter 2.

Example

This example assembles code into .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 A000'          ADD     ACC, @START
15 000007 A003'          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             ****************************************
24             ** Resume assembling into .text section.**
25             ****************************************
26 000008                .text
27 000008 0051           .string "Quit"
   000009 0075
   00000a 0069
   00000b 0074
28
```

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 title of up to 65 characters enclosed in double quotation marks. If you supply more than 65 characters, the assembler truncates the string and issues a warning.

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

Example

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

Source file:

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

Listing file:

```
TMS320C27x Assembler       Version x.xx              Day      Time      Year
Copyright (c) xxxx-xxxx        Texas Instruments Incorporated

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

       2              ;         .
       3              ;         .
       4              ;         .
TMS320C27x Assembler       Version x.xx              Day      Time      Year
Copyright (c) xxxx-xxxx        Texas Instruments Incorporated

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

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

symbol points to the first location reserved by this invocation of the .usect directive. The symbol corresponds to the name of the variable for which you are reserving space.

section name must be enclosed in double quotation marks. This parameter names the uninitialized section. The name can have up to 200 characters. A *section name* can contain a subsection name in the form *section name*:*subsection name*.

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

blocking flag is an optional parameter. If specified and nonzero, the flag means that this section will be blocked. Blocking is an address mechanism similar to alignment, but weaker. It means a section 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 blocking applies to the section, not to the object declared with this instance of the .usect directive.

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

type is an optional parameter. Designating a *type* causes the assembler to produce the appropriate debug information for the *symbol*. See section 3.12, *C-Type Symbolic Debugging for Assembly Variables (–mg option*), on page 3-34 for more information.

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

Variables that can be located contiguously in memory can be defined in the same section; to define variables in this manner, repeat the .usect directive with the same *section name*.

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

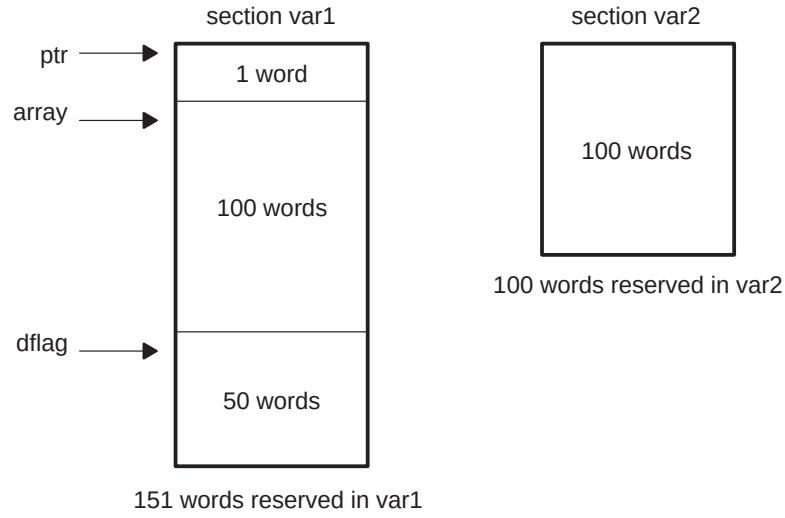
Example

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

```
 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 4–7 on page 4-77 shows how this example reserves space in two uninitialized sections, var1 and var2.

Figure 4–7. The .usect Directive



Chapter 5

Macro Language

The assembler supports a macro language that enables you to create your own instructions. This is especially useful when a program executes a particular task several times. The macro language lets you:

- ❏ Define your own macros and redefine existing macros
- ❏ Simplify long or complicated assembly code
- ❏ Access macro libraries created with the archiver
- ❏ Define conditional and repeatable blocks within a macro
- ❏ Manipulate strings within a macro
- ❏ Control expansion listing

Topic **Page**

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 where you would otherwise repeat the routine. This simplifies and shortens your source program.

If you want to call a macro several times but with different data each time, you can assign parameters within a macro. This enables you to pass different information to the macro each time you call it. The macro language supports a special symbol called a *substitution symbol*, which is used for macro parameters. See section 5.3, *Macro Parameters/Substitution Symbols*, page 5-5, for more information.

Using a macro is a three-step process.

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

- ❏ Macros can be defined at the beginning of a *source file* or in a .copy/.include file. See section 5.2, *Defining Macros*, for more information.
- ❏ Macros can also be defined in a *macro library*. A macro library is a collection of files in archive format created by the archiver. Each member of the archive file (macro library) 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, *Macro Libraries*, page 5-13.

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

Step 3: **Expand the macro.** The assembler expands 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, *Using Directives to Format the Output Listing*, page 5-19.

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

5.2 Defining Macros

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file or in a .copy/.include file (see page 4-31); they can also be defined in a macro library. For more information, see section 5.4, *Macro Libraries*, page 5-13.

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, *Using Recursive and Nested Macros*, page 5-21.

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

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

macname names the macro. You must place the name in the source statement's label field. Only the first 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, *Macro Parameters/Substitution Symbols*, page 5-5.

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 will fail and completion of the macro is unnecessary.

.endm is the directive that terminates the macro definition.

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

Example 5–1. Macro Definition, Call, and Expansion

```
       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
      16

 No Errors, No Warnings
```

If you want to include comments with your macro definition but *do not* want those comments to appear in the macro expansion, use an exclamation point (!) to precede your comments. If you *do* want your comments to appear in the macro expansion, use an asterisk or semicolon. See section 5.7, *Producing Messages in Macros*, page 5-17, for more information about macro comments.

5.3 Macro Parameters/Substitution Symbols

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

Macro parameters are substitution symbols that represent a character string. These symbols can also be used outside of macros to equate a character string to a symbol name (see section 3.7.6, *Substitution Symbols*, page 3-23).

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 in which they are defined. 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, *Substitution Symbols as Local Variables in Macros*, page 5-12.

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

If you pass a list of arguments to one parameter or if you pass a comma or semicolon to a parameter, you must 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.

  The syntax of the .asg directive is:

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

  The quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the substitution symbol.

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

Example 5–3. The .asg Directive

```
   .asg   "A4", RETVAL              ; return value
   .asg   "B14", PAGEPTR            ; global page pointer
   .asg   """Version 1.0""", version
   .asg   "p1, p2, p3", list
```

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

  The syntax of the .eval directive is:

  .eval *well-defined expresssion, substitution symbol*

  The .eval directive evaluates the expression and assigns the string value of the result to the substitution symbol. If the expression is not well defined, the assembler generates an error and assigns the null string to the symbol.

Example 5–4 shows arithmetic being performed on substitution symbols.

Example 5–4. The .eval Directive

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

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

For information about the .asg and .eval assembler directives, see page 4-24.

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 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. Functions and Return Values

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

† For more information about predefined register names, see section 3.7.5, *Predefined Symbolic Constants*, on page 3-22.

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

 No Errors, No Warnings
```

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 as-
sign z = "x"
       3                .asg  "z",y  ; declare y and as-
sign y = "z"
       4                .asg  "y",x  ; declare x and as-
sign x = "y"
       5 000000 FF10      ADD   ACC, x
         000001 0000!
       6
```

5.3.4 Forced Substitutions

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    0
```

The preceding code generates the following source code:

```
  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 the location at which subscripting begins and the length of the resulting character string. *The index of substring characters begins with 1, not 0.*

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

Example 5–8. Using Subscripted Substitution Symbols to Redefine an Instruction

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


ADDX         #100                        ;macro call
```

In Example 5–8, subscripted substitution symbols redefine the STW instruction so that it handles immediates.

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
```

In Example 5–9, the subscripted substitution symbol is used to find a substring strg1 beginning at position start in the string strg2. The position of the substring strg1 is assigned to the substitution symbol pos.

5.3.6 Substitution Symbols as Local Variables in Macros

If you want to use substitution symbols as local variables within a macro, you can use the **.var** directive to define up to 32 local macro substitution symbols (including parameters) per macro. The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and 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 on page 5-11.

5.4 Macro Libraries

One way to define macros is to create 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 on page 4-55). The syntax is:

.mlib ['']*filename*['']

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

The assembler expands the library entry in the same way it expands other macros. (See section 5.1, *Using Macros*, on page 5-2 for information on how the assembler expands macros.) You can control the listing of library entry expansions with the .mlist directive. For more information about the .mlist directive, see section 5.8, *Using Directives to Format the Output Listing*, on page 5-19 and the .mlist description on page 4-57. Only macros that are actually called from the library are extracted, and they are extracted only once.

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

5.5 Using Conditional Assembly in Macros

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

.if *well-defined expression*
[**.elseif** *well-defined expression*]
[**.else**]
.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 the .if expression is false (0), the assembler continues to the code following the .endif directive. For more information on the .if/.elseif/.else/.endif directives, see page 4-46.

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

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

The **.loop** directive's optional *well-defined expression* evaluates to the loop count (the number of loops to be performed). If the expression is omitted, the loop count defaults to 1024 unless the assembler encounters a .break directive with an expression that is true (nonzero). For more information on the .loop/.break/.endloop directives, see page 4-54.

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

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

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

```
      .asg 1,x
      .loop

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

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

Example 5–11. Nested Conditional Assembly Directives

```
       .asg   1,x
       .loop

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

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

Example 5–12. Built-In Substitution Symbol Functions 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

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

For more information, see section 4.8, *Directives That Enable Conditional Assembly*, on page 4-18.

5.6 Using Labels in Macros

All labels in an assembly language program must be unique. This includes labels in macros. If a macro is expanded more than once, its labels are defined more than once. *Defining a label more than once is illegal*. The macro language provides a method of defining labels in macros so that the labels are unique. Simply follow the label with a question mark, and the assembler replaces the question mark with a 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.

Example 5–13. Unique Labels in a Macro

```
       1                    min     .macro  x,y,z
       2
       3                            MV      y,z
       4                    ||      CMPLT   x,y,y
       5                    [y]     B       l?
       6                            NOP     5
       7                            MV      x,z
       8                    l?
       9                            .endm
      10
      11
      12 00000000                   MIN     A0,A1,A2
1
1        00000000 010401A1          MV      A1,A2
1        00000004 00840AF8  ||      CMPLT   A0,A1,A1
1        00000008 80000292  [A1]    B       l?
1        0000000c 00008000          NOP     5
1        00000010 010001A0          MV      A0,A2
1        00000014           l?


LABEL                                 VALUE       DEFN    REF

.TMS320C60                         00000001          0
.tms320C60                         00000001          0
l$1$                               00000014'        12     12
```

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 –x option (see page 3-5).

5.7 Producing Messages in Macros

The macro language supports three directives that enable you to define your own assembly-time error and warning messages. These directives are especially useful when you want to create messages that are 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 it does not set the error count or prevent the creation of an object file.

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

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

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

Example 5–14. Producing Messages in a Macro

```
       1              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
```

For information about the .emsg, .mmsg, and .wmsg assembler directives, see page 4-36.

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 false conditional blocks (blocks that do not generate code). Conditional blocks appear in the listing exactly as they appear in the source code.

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

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

❑ **Substitution-symbol-expansion-listing**

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

.ssnolist turns off substitution-symbol expansion in the listing.

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

❑ **Directive listing**

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

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

For directive listing, .drlist is the default.

5.9 Using Recursive and Nested Macros

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

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

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

Example 5–15. Using Nested Macros

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

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

Example 5–16 shows recursive macros. The fact macro produces assembly code necessary to calculate the factorial of n, where n is an immediate value. The result is placed in the A1 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   temp
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 following directives can be used with macros. The .macro, .mexit, .endm and .var directives are only valid with macros; the remaining directives are general assembly language directives.

Table 5–2. Creating Macros

Mnemonic and Syntax	Description	See Page: Macro Use	See Page: Directive Description
.endm	End macro definition	5-3	5-3
macname **.macro** [*parameter*$_1$] [**,** ... **,** *parameter*$_n$]	Define macro by *macname*	5-3	5-3
.mexit	Go to .endm	5-3	5-3
.mlib *filename*	Identify library containing macro definitions	5-13	4-55

Table 5–3. Manipulating Substitution Symbols

Mnemonic and Syntax	Description	See Page: Macro Use	See Page: Directive Description
.asg [**"**]*character string*[**"**], *substitution symbol*	Assign character string to substitution symbol	5-6	4-24
.eval *well-defined expression, substitution symbol*	Perform arithmetic on numeric substitution symbols	5-7	4-24
.var *sym*$_1$ [,*sym*$_2$, ... ,*sym*$_n$]	Define local macro symbols	5-12	5-12

Table 5–4. Conditional Assembly

Mnemonic and Syntax	Description	See Page: Macro Use	See Page: Directive Description
.break [*well-defined expression*]	Optional repeatable block assembly	5-14	4-54
.endif	End conditional assembly	5-14	4-46
.endloop	End repeatable block assembly	5-14	4-54
.else	Optional conditional assembly block	5-14	4-46
.elseif *well-defined expression*	Optional conditional assembly block	5-14	4-46
.if *well-defined expression*	Begin conditional assembly	5-14	4-46
.loop [*well-defined expression*]	Begin repeatable block assembly	5-14	4-54

Table 5–5. Producing Assembly-Time Messages

Mnemonic and Syntax	Description	See Page: Macro Use	See Page: Directive Description
.emsg	Send error message to standard output	5-17	4-36
.mmsg	Send assembly-time message to standard output	5-17	4-36
.wmsg	Send warning message to standard output	5-17	4-36

Table 5–6. Formatting the Listing

Mnemonic and Syntax	Description	See Page: Macro Use	See Page: Directive Description
.fclist	Allow false conditional code block listing (default)	5-19	4-39
.fcnolist	Suppress false conditional code block listing	5-19	4-39
.mlist	Allow macro listings (default)	5-19	4-57
.mnolist	Suppress macro listings	5-19	4-57
.sslist	Allow expanded substitution symbol listing	5-19	4-67
.ssnolist	Suppress expanded substitution symbol listing (default)	5-19	4-67

Chapter 6

Archiver Description

The TMS320C27x™ 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.

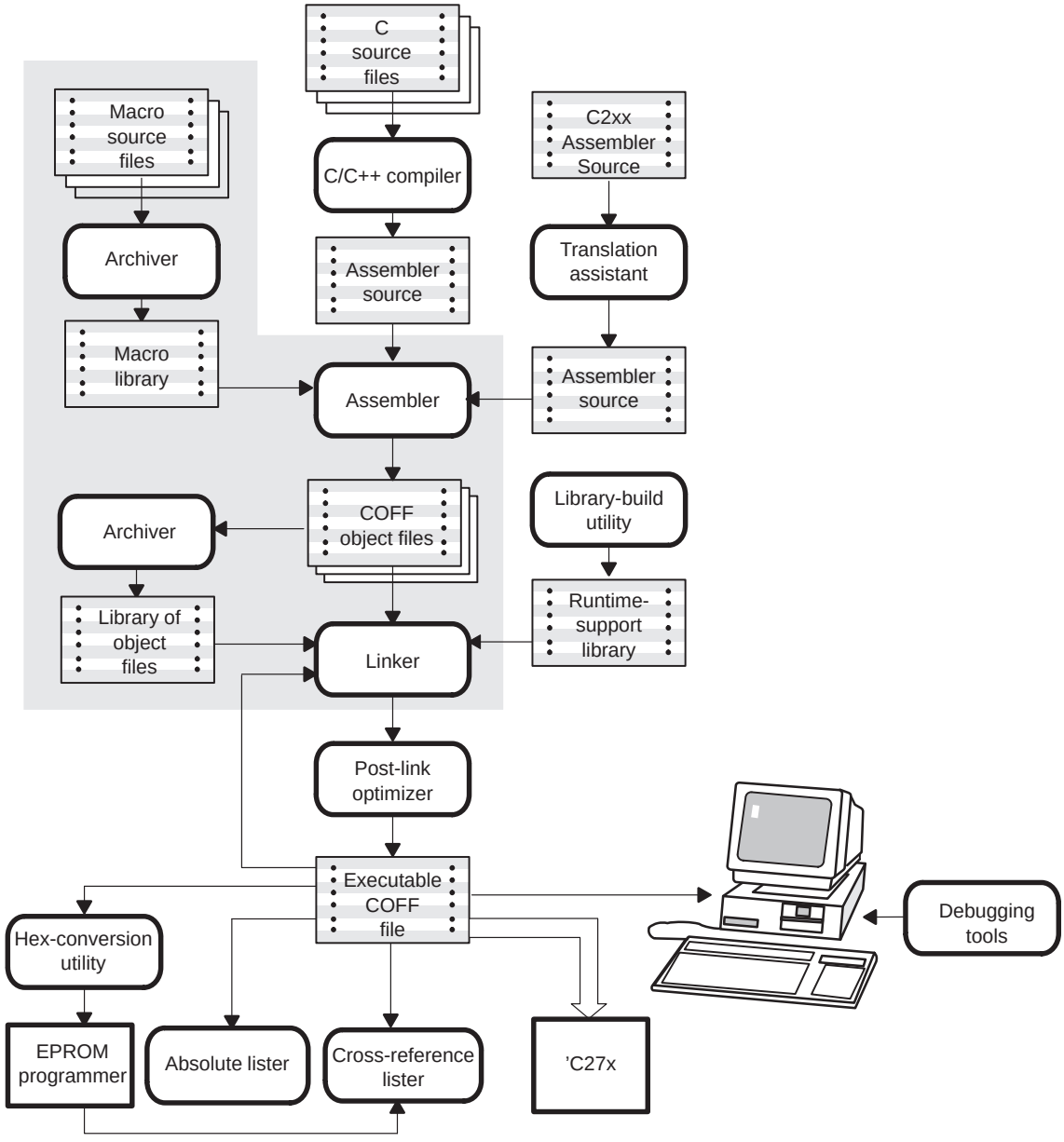
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The 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 the 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 TMS320C27x 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 *filenames*. A command can be preceded by an optional hyphen. You must use one of the following commands when you invoke the archiver, but you can use only one command per invocation. The archiver commands are as follows:

- **@** uses the contents of the specified file instead of command line entries. You can use this command to avoid limitations on command line length imposed by the host operating system. Use a ; at the beginning of a line in the command file to include comments. (See page 6-7 for an example using an archiver command file.)
- **a** adds the specified files to the library. This command does not replace an existing member that has the same name as an added file; it simply *appends* new members to the end of the archive.
- **d** deletes the specified members from the library.
- **r** replaces the specified members in the library. If you do not specify filenames, the archiver replaces the library members with files of the same name in the current directory. If the specified file is not found in the library, the archiver adds it instead of replacing it.
- **t** prints a table of contents of the library. If you specify filenames, only those files are listed. If you do not specify any filenames, the archiver lists all the members in the specified library.
- **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. A *filename* can be up to 15 characters in length; the archiver truncates *filenames* that are longer than 15 characters.

Note: Naming Library Members

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

6.4 Archiver Examples

The following are examples of typical archiver operations:

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

  ar2000 –a function sine.obj cos.obj flt.obj

  The archiver responds as follows:

```
TMS320C27x Archiver            Version x.xx
Copyright (c) xxxx–xxxx Texas Instruments Incorporated
  ==> new archive 'function.lib'
  ==>  building archive 'function.lib'
```

- If you want to print a table of contents of function.lib, use the –t command. Enter:

  ar2000 –t function

  The archiver responds as follows:

```
TMS320C27x Archiver            Version x.xx
Copyright (c) xxxx–xxxx Texas Instruments Incorporated

         FILE NAME    SIZE   DATE
  ---------------   -----   ------------------------
          sine.obj     300   Wed Apr 16 10:00:24 1997
           cos.obj     300   Wed Apr 16 10:00:30 1997
           flt.obj     300   Wed Apr 16 09:59:56 1997
```

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

  ar2000 –as function atan.obj

  The archiver responds as follows:

```
TMS320C27x Archiver            Version x.xx
Copyright (c) xxxx–xxxx Texas Instruments Incorporated
  ==> symbol defined: '_sin'
  ==> symbol defined: '$sin'
  ==> symbol defined: '_cos'
  ==> symbol defined: '$cos'
  ==> symbol defined: '_tan'
  ==> symbol defined: '$tan'
  ==> symbol defined: '_atan
  ==> symbol defined: '$atan'
  ==> building archive 'function.lib'
```

  Because this example doesn't 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.obj

  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.obj

- 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:

```
TMS320C27x Archiver            Version x.xx
Copyright (c) xxxx–xxxx Texas Instruments Incorporated
  ==>  building archive 'modules.lib'
```

  This is the modules.cmd command file:

```
; Command file to replace members of the
;     modules library with updated files
; 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
```

  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.

Chapter 7

Linker Description

The TMS320C27x™ linker creates executable modules by combining COFF object files. The concept of COFF sections is basic to linker operation; Chapter 2, *Introduction to Common Object File Format*, discusses the COFF format in detail.

Topic **Page**

7.1 Linker Overview

The TMS320C27x linker allows you to configure system memory by allocating output sections efficiently into the memory map. As the linker combines object files, it performs the following tasks:

- ❏ Allocates sections into the target system's configured memory
- ❏ Relocates symbols and sections to assign them to final addresses
- ❏ Resolves undefined external references between input files

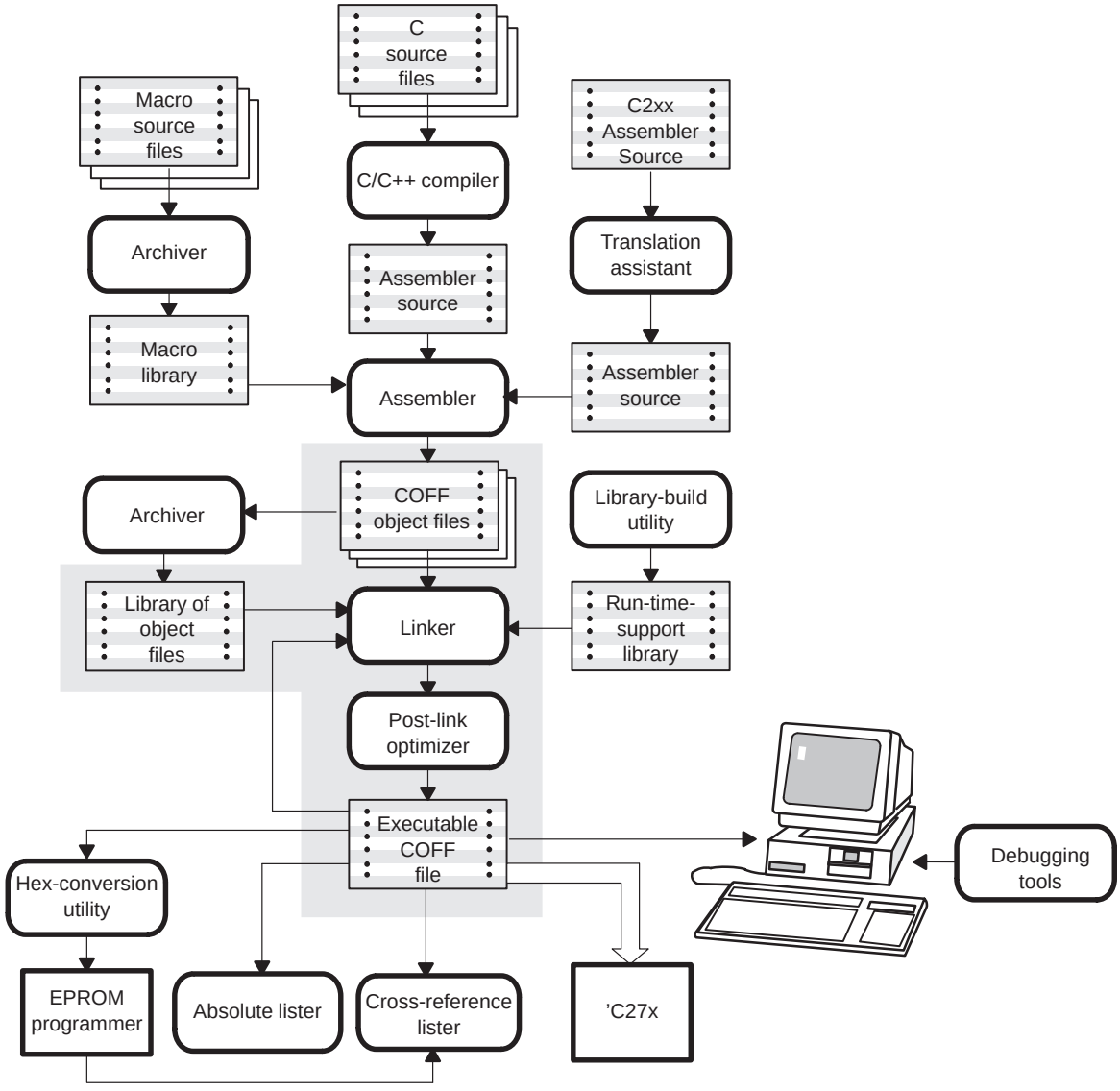
The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow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

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

7.2 The Linker's Role in the Software Development Flow

Figure 7–1 illustrates the linker's role in the software development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable COFF object module that can be downloaded to one of several development tools or executed by a TMS320C27x device.

Figure 7–1. The Linker in the TMS320C27x Software Development Flow



7.3 Invoking the Linker

The general syntax for invoking the linker is:

lnk2000 [*options*] *filename*$_1$, ... [*filename*$_n$]

lnk2000 is the command that invokes the linker.

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

filenames can be object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the –o option to name the output file.

There are three methods for invoking the linker:

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

  ```
  lnk2000 file1.obj file2.obj –o link.out
  ```

- Enter the **lnk2000** command with no filenames or options; the linker prompts for them:

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

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

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

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

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

  lnk2000 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:

  lnk2000 –m link.map linker.cmd file3.obj

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

7.4 Linker Options

Linker options control linking operations. They can be placed on the command line or in a command file. Linker options must be preceded by a hyphen (–). The order in which options are specified is unimportant, except for the –l and –i options. Since the –l option reads from the –i option(s), the –i option(s) must appear before the –l option. Options can be separated from arguments (if they have them) by an optional space. Table 7–1 summarizes the linker options.

You can string together the options that do not have parameters (for example, lnk2000 –ar) or enter them separately (for example, lnk2000 –a –r). You must specify options that have parameters separately from other options (for example, lnk2000 –i dsptools –ar).

Table 7–1. Linker Options Summary

Option	Description	Page
–a	Produces an absolute, executable module. This is the default; if neither –a nor –r is specified, the linker acts as if –a were specified.	7-8
–ar	Produces a relocatable, executable object module	7-8
–b	Disables merge of symbolic debugging information	7-9
–c	Autoinitializes variables at run time	7-10
–cr	Autoinitializes variables at load time	7-10
–e *global_symbol*	Defines a global *symbol* that specifies the primary entry point for the output module	7-10
–f *fill_value*	Sets the default fill value for holes within output sections; *fill_value* is a 16-bit constant	7-11
–g *symbol*	Makes *symbol* global (overrides –h)	7-11
–h	Makes all global symbols static	7-11
–heap *size*	Sets heap (for the dynamic memory allocation in C) size to *size* words and defines a global symbol that specifies the heap size. The default size is 1K words	7-12
–i *pathname* †	Alters the library-search algorithm to look in a directory named with *pathname* before looking in the default location. This option must appear before the –l option	7-13
–j	Disables conditional linking	7-15
–k	Forces the linker to ignore any SECTIONS directive alignment specification	7-15
–l *filename* †	Names an archive library or linker command file *filename* as linker input	7-12
–m *filename* †	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename*	7-16
–o *filename* †	Names the executable, output module. The default filename is a.out	7-17
–q	Suppresses the banner and all progress information (quiet)	7-17
–r	Produces a nonexecutable, relocatable output module	7-8
–s	Strips symbol-table information and line-number entries from the output module	7-18
–stack *size*	Sets C system stack size to *size* words and defines a global symbol that specifies the stack size. The default size is 1K words	7-18
–u *symbol*	Places an unresolved external *symbol* into the output module's symbol table	7-18
–w	Displays a message when an undefined output section is created	7-19
–x	Forces rereading of libraries, which resolves back references	7-20

† The *pathname* or *filename* must follow operating-system conventions.
‡ For more information, see the *TMS320C27x Optimizing C/C++ Compiler User's Guide.*

7.4.1 Relocation Capabilities (–a and –r Options)

The linker performs relocation, which is the process of adjusting all references to a symbol when the symbol's address changes. The linker supports two options (–a and –r) that allow you to produce an absolute or relocatable output module.

❑ **Producing an absolute output module (–a option)**

When you use the –a option without the –r option, the linker produces an absolute, executable output module. Absolute files contain *no* relocation information. Executable files contain the following:

- Special symbols defined by the linker (section 7.14.4 on page 7-58)
- 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:

```
lnk2000 -a file1.obj file2.obj
```

Note: –a and –r Options

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

❑ **Producing a relocatable output module (–r option)**

When you use the –r option without the –a option, the linker retains relocation entries in the output module. If the output module will be relocated (at load time) or relinked (by another linker execution), use –r to retain the relocation entries.

The linker produces a file that is not executable when you use the –r option without –a. A file that is not executable does not contain special linker symbols or an optional header. The file may contain unresolved references, but these references do not prevent creation of an output module.

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

```
lnk2000 -r file1.obj file2.obj
```

The output file a.out can be relinked with other object files or relocated at load time. (Linking a file that will be relinked with other files is called partial linking. For more information, see section 7.16, *Partial (Incremental) Linking*, on page 7-63.)

❑ **Producing an executable, relocatable output module (–ar option combination)**

If you invoke the linker with both the –a and –r options, the linker produces an *executable, relocatable* object module. The output file contains the special linker symbols, an optional header, and all resolved symbol references, but the relocation information is retained.

This example links file1.obj and file2.obj and creates an executable, relocatable output module called xr.out:

```
lnk2000 –ar file1.obj file2.obj –o xr.out
```

When the linker encounters a file that contains no relocation or symbol-table information, it issues a warning message (but continues executing). Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated (that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

7.4.2 Disable Merge of Symbolic Debugging Information (–b Option)

By default, the linker eliminates duplicate entries of symbolic debugging information. Such duplicate information is commonly generated when a C program is compiled for debugging. For example:

```
–[ header.h ]–
typedef struct
{
   <define some structure members>
} XYZ;

–[ f1.c ]–
#include ”header.h”
...

–[ f2.c ]–
#include ”header.h”
...
```

When these files are compiled for debugging, both f1.obj and f2.obj have symbolic debugging entries to describe type XYZ. For the final output file, only one set of these entries is necessary. The linker merges the duplicate entries automatically.

To disable the merge of duplicate entries, use the –b linker command option. The linker runs faster and uses less machine memory when using –b. The –b option should appear before object files, or it will be ignored by the linker.

7.4.3 C Language Options (–c and –cr Options)

The –c and –cr options cause the linker to use linking conventions that are required by the C/C++ compiler.

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

For more information, see section 7.17, *Linking C Code*, on page 7-65; section 7.17.4, *Autoinitialization of Variables at Run Time*, on page 7-66; and section 7.17.5, *Autoinitialization of Variables at Load Time*, on page 7-67.

7.4.4 Define an Entry Point (–e Option)

The memory address at which a program begins executing is called the *entry point*. When a loader loads a program into target memory, the program counter (PC) must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table.

1) The value specified by the –e option. The syntax is:

   –e *global_symbol*

   where *global_symbol* defines the entry point and must appear as an external symbol in one of the input files.

2) The value of symbol _c_int00 (if present). The _c_int00 symbol *must* be the entry point if you are linking code produced by the C/C++ compiler.

3) The value of symbol _main (if present)

4) 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.

```
lnk2000 –e begin file1.obj file2.obj
```

7.4.5 Set Default Fill Value (–f *fill_value* Option)

The –f option fills the holes formed within output sections or initializes uninitialized sections when they are combined with initialized sections. This allows you to initialize memory areas during link time without reassembling a source file. The argument *fill_value* is a 16-bit constant (up to four hexadecimal digits). If you do not use –f, the linker uses 0 as the default fill value.

This example fills holes with the hexadecimal value ABCD:

```
lnk2000 -f 0ABCDh file1.obj file2.obj
```

For more information on creating and filling holes, see section 7.15, *Creating and Filling Holes*, on page 7-59.

7.4.6 Make a Symbol Global (–g *symbol* Option)

The –h option makes all global symbols static. If you have a symbol that you want to remain global and you use the –h option, you can use the –g option to declare that symbol to be global. The –g option overrides the effect of the –h option for the symbol that you specify. The syntax for the –g option is:

–g *symbol*

7.4.7 Make All Global Symbols Static (–h Option)

The –h option makes all global symbols static. Static symbols are not visible to externally linked modules. By making global symbols static, global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.

The –h option effectively nullifies all .global assembler directives. All symbols become local to the module in which they are defined, so no external references are possible. For example, assume file1.obj and file2.obj both define global symbols called EXT. By using the –h option, you can link these files without conflict. The symbol EXT defined in file1.obj is treated separately from the symbol EXT defined in file2.obj.

```
lnk2000 -h file1.obj file2.obj
```

7.4.8 Define Heap Size (–heap *size* Option)

The C/C++ compiler uses an uninitialized section called .sysmem for the C memory pool used by malloc(). You can set the size of this memory pool at link time by using the –heap option. Specify the *size* as a constant immediately after the option. The example below creates a heap that is 2K words in size:

```
lnk2000 –heap 0x0800  /* defines a 2k heap (.sysmem section)*/
```

The linker creates the .sysmem section only if there is a .sysmem section in an input file.

The linker also creates a global symbol __SYSMEM_SIZE and assigns it a value equal to the size of the heap. The default size is 1K words.

For more information, see section 7.17, *Linking C Code*, on page 7-65.

7.4.9 Alter the Library Search Algorithm (–l Option, –i Option, and C_DIR Environment Variable)

Usually, when you want to specify a library or linker command file as linker input, you simply enter the library or command filename as you would any other input filename; the linker looks for the filename 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:

```
lnk2000 file1.obj object.lib
```

If you want to use a library or command file that is not in the current directory, use the –l (lowercase L) linker option. The syntax for this option is:

–l *filename*

The *filename* is the name of an archive library or linker command file; the space between –l and the filename is optional.

You can augment the linker's directory search algorithm by using the –i linker option or the environment variable. The linker searches for object libraries specified by the –l option in the following order:

1) It searches directories named with the –i linker option. The –i option must appear before the –l option on the command line or in a command file.

2) It searches directories named with C_DIR.

3) If C_DIR is not set, it searches directories named with the assembler's A_DIR environment variable.

4) It searches the current directory.

7.4.9.1 Name an Alternate Library Directory (–i pathname Option)

The –i option names an alternate directory that contains object libraries. The syntax for this option is:

–i *pathname*

The *pathname* names a directory that contains object libraries; the space between –i and the *pathname* is optional.

When the linker is searching for object libraries named with the –l option, it searches through directories named with –i first. Each –i option specifies only one directory, but you can use several –i options per invocation. When you use the –i option to name an alternate directory, it must precede any –l option used on the command line or in a command file.

For example, assume that there are two archive libraries called r.lib and lib2.lib. Assume the following paths for the libraries:

DOS or OS/2	c:\ld\r.lib and c:\ld2\lib2.lib
UNIX:	/ld/r.lib and /ld2/lib2.lib
Windows:	c:\ld\r.lib and c:\ld2\lib2.lib

The following examples show how you can set the –i option and use both libraries during a link:

Operating System	Enter
DOS or OS/2	`lnk6x f1.obj f2.obj –i\ld –i\ld2 –lr.lib –llib2.lib`
UNIX	`lnk2000 f1.obj f2.obj –i/ld –i/ld2 –lr.lib –llib2.lib`
Windows	`lnk2000 f1.obj f2.obj –i\ld –i\ld2 –lr.lib –llib2.lib`

7.4.9.2 Name an Alternate Library Directory (C_DIR Environment Variable)

An environment variable is a system symbol that you define and assign a string to. The linker uses an environment variable named C_DIR to name alternate directories that contain object libraries. The command syntaxes for assigning the environment variable are:

Operating System	Enter
DOS or O/2	**set C_DIR=** *pathname$_1$;pathname$_2$; . . .*
UNIX	**setenv C_DIR "***pathname$_1$;pathname$_2$; . . .***"**
Windows	**set C_DIR=** *pathname$_1$;pathname$_2$; . . .*

The *pathnames* are directories that contain object libraries. Use the –l (lowercase L) linker option on the command line or in a command file to tell the linker to search these directories for the libraries.

For example, assume that there are two archive libraries called r.lib and lib2.lib. Assume the following paths for the library files:

DOS or OS/2	c:\ld\r.lib and c:\ld2\lib2.lib
UNIX:	/ld/r.lib and /ld2/lib2.lib
Windows:	c:\ld\r.lib and c:\ld2\lib2.lib

The following examples show how to set the environment variable and use both libraries during a link.

Operating System	Enter
DOS or OS/2	**set C_DIR=\ld;\ld2** **lnk6x f1.obj f2.obj –l r.lib –l lib2.lib**
UNIX	**setenv C_DIR "/ld ;/ld2"** **lnk2000 f1.obj f2.obj –l r.lib –l lib2.lib**
Windows	**set C_DIR=\ld;\ld2** **lnk2000 f1.obj f2.obj –l r.lib –l lib2.lib**

The environment variable remains set until you reboot the system or reset the variable by entering:

Operating System	Enter
DOS or OS/2	**set C_DIR=**
UNIX	**unsetenv C_DIR**
Windows	**set C_DIR=**

The assembler uses an environment variable named A_DIR to name alternate directories that contain .copy/.include files or macro libraries. If C_DIR is not set, the linker searches for object libraries in the directories named with A_DIR. For more information about object libraries, see section 7.6 on page 7-24.

7.4.10 Disable Conditional Linking (–j Option)

The –j option disables conditional linking that has been set up with the assembler .clink directive. By default, all sections are unconditionally linked. See page 4-30 for information on setting up conditional linking using the .clink directive.

7.4.11 Ignore Alignment (–k Option)

The –k option forces the linker to ignore all alignments specified with the .align assembler directive. For more information about the .align directive, see page 4-23.

7.4.12 Create a Map File (–m *filename* Option)

The –m option creates a linker map listing and puts it in *filename.* The syntax for the –m option is:

–m *filename*

The linker map designates:

- ❏ 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). This information is generated on the basis of the information in the MEMORY directive in the linker command file. The table has the following columns;:
  - ■ **Name.** This is the name of the memory range specified with the MEMORY directive.
  - ■ **Origin.** This specifies the starting address of a memory range.
  - ■ **Length.** This specifies the length of a memory range.
  - ■ **Attributes.** This specifies 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.
  - ■ **Fill.** This specifies a fill character for the memory range.

  For more information about the MEMORY directive, see section 7.7, *The MEMORY Directive*, on page 7-26.
- ❏ A table showing the linked addresses of each output section and the input sections that make up the output sections (section allocation map). This information is generated on the basis of the information in the SECTIONS directive in the linker command file. The table has the following columns:
  - ■ **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.

For more information about the SECTIONS directive, see section 7.8, *The SECTIONS Directive*, on page 7-31.

❑ Two tables showing external symbols and their address, one sorted by name, and one sorted by address.

This example links file1.obj and file2.obj and creates a map file called map.out:

```
lnk2000 file1.obj file2.obj -m map.out
```

Example 7–14 on page 7-71 shows an example of a map file.

7.4.13 Name an Output Module (–o *filename* Option)

The linker creates an output module when no errors are encountered. If you do not specify a filename for the output module, the linker gives it the default name a.out. If you want to write the output module to a different file, use the –o option. The syntax for the –o option is:

–o *filename*

The *filename* is the new output module name.

This example links file1.obj and file2.obj and creates an output module named run.out:

```
lnk2000 -o run.out file1.obj file2.obj
```

7.4.14 Specify a Quiet Run (–q Option)

The –q option suppresses the linker's banner, but it must be the first option listed. If it is not, the banner is displayed. This option is useful for batch operation.

7.4.15 Strip Symbolic Information (–s Option)

The –s option creates a smaller output module by omitting symbol-table information and line number entries. The –s option is useful for production applications when you must create the smallest possible output module.

This example links file1.obj and file2.obj and creates an output module that is stripped of line numbers and symbol-table information, named nosym.out.

```
lnk2000 –o nosym.out -s file1.obj file2.obj
```

Because the –s option strips symbolic information from the output module, using the –s option limits later use of a symbolic debugger and prevents a file from being relinked.

7.4.16 Define Stack Size (–stack *size* Option)

The C/C++ compiler uses an uninitialized section, .stack, to allocate space for the run-time stack. You can set the size of this section at link time with the –stack option. Specify the size as a constant immediately after the option. The example below creates a stack that is 4K words in size:

```
lnk2000 -stack 0x1000  /* defines a 4K stack (.stack
section) */
```

If you specify a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size is different.

When the linker defines the .stack section, it also defines a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the section. The default software stack size is 1K words.

7.4.17 Introduce an Unresolved Symbol (–u *symbol* Option)

The –u option introduces an unresolved symbol into the linker's symbol table. This forces the linker to search a library and include the member that defines the symbol. The linker must encounter the –u option *before* it links in the member that defines the symbol. The syntax for the –u option is:

–u *symbol*

For example, suppose a library named rts.lib contains a member that defines the symbol symtab; none of the object files being linked reference symtab. However, suppose you plan to relink the output module and you would like to include the library member that defines symtab in this link. Using the –u option as shown in the example that follows forces the linker to search rts.lib for the member that defines symtab and to link in the member.

```
lnk2000 -u symtab file1.obj file2.obj rts.lib
```

If you do not use –u, this member is not included, because there is no explicit reference to it in file1.obj or file2.obj.

7.4.18 Display a Message When an Undefined Output Section Is Created (–w Option)

The –w linker command option displays additional messages pertaining to the creation of memory sections. Additional messages are displayed in the following circumstances:

- In a linker command file, you can set up a SECTIONS directive that describes how input sections are combined into output sections. However, if the linker encounters input sections that do not have a corresponding output section defined in the SECTIONS directive, the linker combines the input sections that have the same name into an output section with that name. By default, the linker does not display a message to tell you when this has occurred.

  If this situation occurs and you use the –w option, the linker displays a message when it creates a new output section.

- If you are linking C code and you do not use the –heap and –stack options, the linker creates the .sysmem and .stack sections, respectively, for you. Each section has a default size of 1K words. You might not have enough memory available for one or both of these sections. In this case, the linker issues an error message saying a section could not be allocated.

  If you use the –w option, the linker displays another message with more information including which directive to use to allocate the .sysmem or .stack section yourself.

For more information about the SECTIONS directive, see section 7.8 on page 7-31. For more information about the default actions of the linker, see section 7.13 on page 7-53.

7.4.19 Exhaustively Read Libraries (–x Option)

The linker normally reads input files, including archive libraries, only once: when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library (this is called a *back reference*), the reference is not resolved.

With the –x option, you can force the linker to repeatedly reread all libraries. The linker continues to reread libraries until no more references can be resolved. 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:

```
lnk2000 –la.lib –lb.lib –la.lib
```

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

```
lnk2000 –x –la.lib –lb.lib
```

Linking with the –x option may be slower than reading each input file once, so you should use it only as needed.

7.5 Linker Command Files

Linker command files allow you to put linking information in a file; this is useful when you invoke the linker often with the same information. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line.

Linker command files are ASCII files that contain one or more of the following items:

- ❏ Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)
- ❏ Linker options, which can be used in the command file in the same manner that they are used on the command line
- ❏ The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration (see section 7.7, *The MEMORY Directive*, on page 7-26). The SECTIONS directive controls how sections are built and allocated (see section 7.8, *The SECTIONS Directive*, on page 7-31.)
- ❏ Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the lnk2000 command and follow it with the name of the command file:

lnk2000 *command_filename*

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links the file. If the linker recognizes the file as an object library, it uses that library to resolve any unresolved references. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it. Command filenames are case sensitive, regardless of the system used.

Example 7–1 shows a sample linker command file called link.cmd.

Example 7–1. Linker Command File

```
a.obj           /*  First input filename          */
b.obj           /*  Second input filename         */
–o prog.out     /*  Option to specify output file */
–m prog.map     /*  Option to specify map file    */
```

The sample file in Example 7–1 contains only filenames and options. (You can place comments in a command file by delimiting them with /* and */.) To invoke the linker with this command file, enter:

```
lnk2000 link.cmd
```

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

```
lnk2000 –r link.cmd c.obj d.obj
```

The linker processes the command file as soon as it encounters it, so a.obj and b.obj are linked into the output module before c.obj and d.obj.

You can specify multiple command files. If, for example, you have a file called names.lst that contains filenames and another file called dir.cmd that contains linker directives, you could enter:

```
lnk2000 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. Example 7–2 shows a sample command file that contains linker directives. For more information about the MEMORY directive, see section 7.7, *The MEMORY Directive*, on page 7-26. For more information about the SECTIONS directive, see section 7.8, *The SECTIONS Directive*, on page 7-31.

Example 7–2. Command File With Linker Directives

```
a.obj b.obj c.obj           /* Input filenames       */
–o prog.out –m prog.map     /* Options               */

MEMORY                      /* MEMORY directive      */
{
  RAM:  origin = 0100h      length = 0100h
  ROM:  origin = 01000h    length = 0100h
}

SECTIONS                    /* SECTIONS directive    */
{
  .text:  > ROM
  .data:  > ROM
  .bss:   > RAM
}
```

7.5.1 Reserved Names in Linker Command Files

The following names are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

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

7.5.2 Constants in Linker Command Files

The linker uses the same syntax for constants that the assembler uses except that it does not support the binary format. See section 3.5, *Constants*, on page 3-11 for a complete description.

7.6 Object Libraries

An object library is a partitioned archive file that contains object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as linker input, the linker includes any members of the library that define existing unresolved symbol references. You can use the archiver to build and maintain libraries. Chapter 6, *Archiver Description*, contains more information about the archiver.

Using object libraries can reduce link time and the size of the executable module. Normally, if an object file that contains a function is specified at link time, the file is linked whether the function is used or not; however, if that same function is placed in an archive library, the file is included only if the function is referenced.

The order in which libraries are specified is important, because the linker includes only those members that resolve symbols that are undefined at the time the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. Alternatively, the –x option can be used to reread libraries until no more references can be resolved (see section 7.4.19, *Exhaustively Read Libraries (–x Option)*, on page 7-20). A library has a table that lists all external symbols defined in the library; the linker searches through the table until it determines that it cannot use the library to resolve any more references.

The following examples link several files and libraries, using these assumptions:

- ❏ Input files f1.obj and f2.obj both reference an external function named clrscr.
- ❏ Input file f1.obj references the symbol origin.
- ❏ Input file f2.obj references the symbol fillclr.
- ❏ Member 0 of library libc.lib contains a definition of origin.
- ❏ Member 3 of library liba.lib contains a definition of fillclr.
- ❏ Member 1 of both libraries defines clrscr.

If you enter:

```
lnk2000 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 (liba.lib) 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:

```
lnk2000 f1.obj f2.obj libc.lib liba.lib
```

then the references to clrscr are satisfied by member 1 of libc.lib.

If none of the linked files reference symbols defined in a library, you can use the –u option to force the linker to include a library member. (See section 7.4.17, *Introduce an Unresolved Symbol (–u* symbol *Option)*, on page 7-18.) The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
lnk2000 –u rout1 libc.lib
```

If any member of libc.lib defines rout1, the linker includes that member.

It is not possible to control the allocation of individual library members; members are allocated according to the SECTIONS directive default allocation algorithm. For more information, see section 7.8, *The SECTIONS Directive*, on page 7-31.

Section 7.4.9, *Alter the Library Search Algorithm (–l Option, –i Option, and C_DIR Environment Variable)*, on page 7-12 describes methods for specifying directories that contain object libraries.

7.7 The MEMORY Directive

The linker determines where output sections are allocated into memory; it must have a model of target memory to accomplish this. The MEMORY directive allows you to specify a model of target memory so that you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections, and it uses the model to determine which memory locations can be used for object code.

The memory configurations of TMS320C27x systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use MEMORY to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

For more information, see section 2.3, *How the Linker Handles Sections*, on page 2-11, and section 2.4, *Relocation*, on page 2-14.

7.7.1 Default Memory Model

If you do not use the MEMORY directive, the linker uses a default memory model that is based on the TMS320C27x architecture. For more information about the default memory model, see section 7.13, *Default Allocation Algorithm*, on page 7-53.

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

TMS320C27x 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, *Overlaying Pages*, on page 7-48.

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 TMS320C27x supports up to 255 address spaces, but the number of address spaces available depends on the customized configuration of your device (see the *TMS320C27x 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 linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in Example 7–3 defines a system that has 4K words of ROM at address 0C00h in program memory, 32 words of RAM at address 60h in data memory, and 512 words at address 200h in data memory.

Example 7–3. The MEMORY Directive

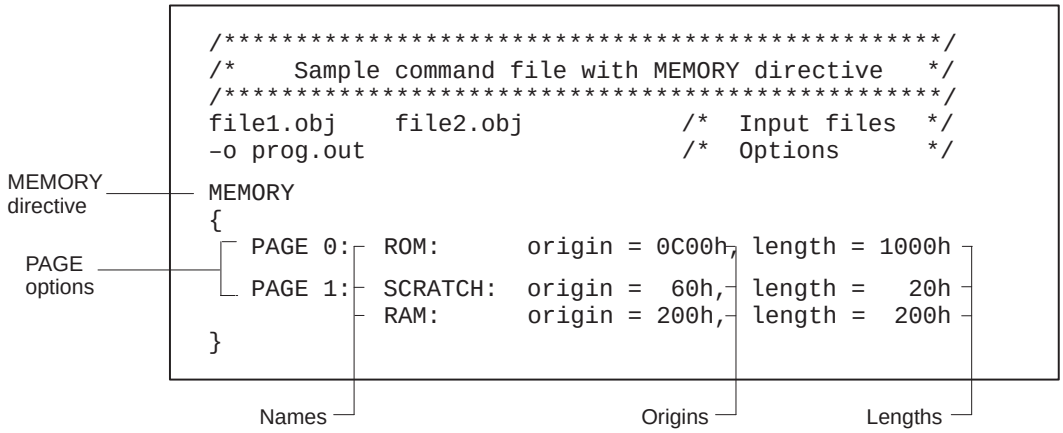
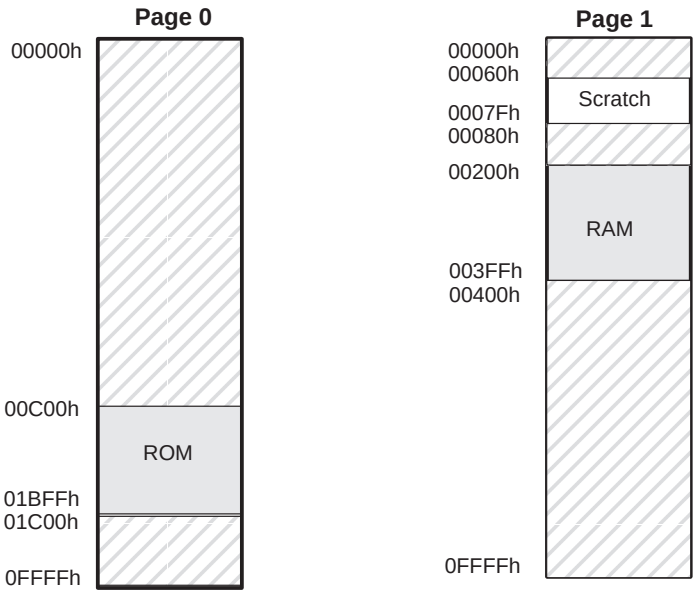


Figure 7–2 illustrates the memory map described in Example 7–3.

Figure 7–2. Memory Map Defined in Example 7–3



You can use the SECTIONS directive to tell the linker where to link the sections. For more information, see section 7.8, *The SECTIONS Directive*, on page 7-31.

The general syntax for the MEMORY directive is:

```
MEMORY
{
    [PAGE 0 : ]  name [(attr)]  :  origin = constant,  length = constant[, fill = constant];
    [PAGE 1 : ]  name [(attr)]  :  origin = constant,  length = constant[, fill = constant];
        .
        .
        .
    [PAGE n : ]  name [(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 eight characters; valid characters include A–Z, a–z, $, ., and _. The names have no special significance to the linker; they simply identify memory ranges. Memory range names are internal to the linker and are not retained in the output file or in the symbol table. All memory ranges must have unique names and must not overlap.

attr specifies one to four attributes associated with the named range. Attributes are optional; when used, they must be enclosed in parentheses. Attributes restrict the allocation of output sections into certain memory ranges. If you do not use any attributes, you can allocate any output section into any range with no restrictions. Any memory for which no attributes are specified (including all memory in the default model) has all four attributes. Valid attributes include:

R specifies that the memory can be read.
W specifies that the memory can be written to.
X specifies that the memory can contain executable code.
I specifies that the memory can be initialized.

origin specifies the starting address of a memory range; enter as *origin*, *org*, or *o*. The value, specified in bytes, is a 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 16-bit integer constant and can be decimal, octal, or hexadecimal. The fill value is used to fill areas of the memory range that are not allocated to a section.

Note: Filling Memory Ranges

If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFh:

```
MEMORY
{
   RFILE (RW) : o = 02h, l = 0FEh, f = 0FFFFh
}
```

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.

7.8 The SECTIONS Directive

The SECTIONS directive:

- Describes how input sections are combined into output sections
- Defines output sections in the executable program
- Specifies where output sections are placed in memory (in relation to each other and to the entire memory space)
- Permits renaming of output sections

For more information, see section 2.3, *How the Linker Handles Sections*, on page 2-11; section 2.4, *Relocation*, on page 2-14; and section 2.2.4, *Subsections*, on page 2-7. Subsections allow you to manipulate sections with greater precision.

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. section 7.13, *Default Allocation Algorithm*, on page 7-53 describes this algorithm in detail.

7.8.1 SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

```
SECTIONS
{
    name : [property [, property] [, property] . . . ]
    name : [property [, property] [, property] . . . ]
    name : [property [, property] [, property] . . . ]
}
```

Each section specification, beginning with *name*, defines an output section. (An output section is a section in the output file.) A section name can be a subsection specification. After the section name is a list of properties that define the section's contents and how the section is allocated. The properties can be separated by optional commas. Possible properties for a section are:

- **Load allocation** defines where in memory the section is to be loaded.

  Syntax:

  load = *allocation* or
  load > *allocation* or
  > *allocation*

  Allocation represents portions of the syntax that specify how sections are placed in the target memory.

- **Run allocation** defines where in memory the section is to be run.

  Syntax:

  run = *allocation* or
  run > *allocation*

- **Input sections** defines the input sections (object files) that constitute the output section.

  Syntax:

  { *input_sections* **}**

- **Section type** defines flags for special section types.

  Syntax:

  type = COPY or
  type = DSECT or
  type = NOLOAD

  For more information, see section 7.12, *Special Section Types (DSECT, COPY, and NOLOAD)*, on page 7-52.

- **Fill value** defines the value used to fill uninitialized holes.

  Syntax:

  fill = *value* or
  name : [*properties*] **=** *value*

  For more information, see section 7.15, *Creating and Filling Holes*, on page 7-59.

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

Example 7–4. The SECTIONS Directive

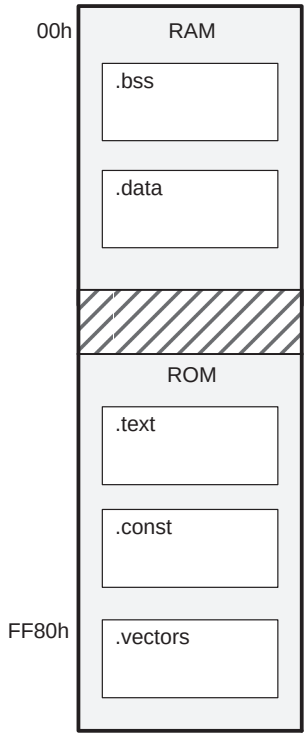
SECTIONS directive

Section specifications

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

SECTIONS
{
  .text:     load = ROM, run = 800h
  .const:    load = ROM
  .bss:      load = RAM
  .vectors:  load = 0FF80h
     {
        t1.obj(.intvec1)
        t2.obj(.intvec2)
        endvec = .;
     }
  .data:     align = 16
}
```

Figure 7–3 shows the five output sections defined by the SECTIONS directive in Example 7–4: .vectors, .text, .const, .bss, and .data.

Figure 7–3. Section Allocation Defined in Example 7–4



The .bss section combines the .bss sections from file1.obj and file2.obj.

The .data section combines the .data sections from file1.obj and file2.obj. The linker places it anywhere space for it is available (in RAM in this illustration) and aligns it to a 16-word boundary.

The .text section combines the .text sections from file1.obj and file2.obj. The linker combines all sections named .text into this section. The application must relocate the section to run at 0800h.

The .const section combines the .const sections from file1.obj and file2.obj.

The .vectors section is composed of the .intvec1 section from t1.obj and the .intvec2 section from t2.obj.

7.8.2 Allocation

The linker assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. The process of locating the output section in the target's memory and assigning its address(es) is called allocation. For more information about using separate load and run allocation, see section 7.9, *Specifying a Section's Run-Time Address*, on page 7-41.

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to allocate the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run allocation are separate, all parameters following the keyword LOAD apply to load allocation, and those following the keyword RUN apply to run allocation. Possible allocation parameters are:

Binding allocates a section at a specific address.

```
.text: load = 0x1000
```

Named memory allocates the section into a range defined in the MEMORY directive with the specified name or attributes.

```
.text: load > ROM
```

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

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

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

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

Page specifies the memory page to be used.

```
.text: load = OVR_MEM PAGE 1
```

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

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

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

```
.text: > ROM align 16 PAGE 2
```

Or if you prefer, use parentheses for readability:

```
.text: load = (ROM align(16) page (2))
```

You can also use an input section specification to identify the sections from input files that are combined to form an output section. For more information, see section 7.8.3, *Specifying Input Sections*, on page 7-39.

The following sections describe these allocation parameters.

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: 0x1000
```

This example specifies that the .text section must begin at location 0x1000. The binding address must be a 22-bit constant.

Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

Note: Binding Is Incompatible With Alignment and Named Memory

You cannot bind a section to an address if you use alignment or named memory. If you try to do this, the linker issues an error message.

7.8.2.2 Named Memory

You can allocate a section into a memory range that is defined by the MEMORY directive (see section 7.7, *The MEMORY Directive*, on page 7-26). This example names ranges and links sections into them:

```
MEMORY
{
   ROM (RIX)  :  origin = 0C00h,  length = 1000h
   RAM (RWIX) :  origin =  80h,  length = 1000h
}

SECTIONS
{
   .text  :   > ROM
   .data  :   > RAM
   .bss   :   > RAM
}
```

In this example, the linker places .text into the area called ROM. The .data and .bss output sections are allocated into RAM.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
   .text:  > (X)   /* .text --> executable memory      */
   .data:  > (RI)  /* .data --> read or init memory    */
   .bss :  > (RW)  /* .bss --> read or write memory    */
}
```

In this example, the .text output section can be linked into either the ROM or RAM area because both areas have the X attribute. The .data section can also go into either ROM or RAM because both areas have the R and I attributes. The .bss output section, however, must go into the RAM area because only RAM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no conflicting assignments exist, the .text section starts at address 0. If a section must start on a specific address, use binding instead of named memory.

7.8.2.3 Alignment and Blocking

You can tell the linker to place an output section at an address that falls on an n-word boundary, where n is a power of 2, by using the *align* keyword. For example:

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

allocates .text so that it falls on a 128-word boundary.

You can also align a section within a named memory range. For example:

```
.data align(128) :   > RAM
```

In this example, the .data section is aligned on a 128-word boundary within the RAM range.

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:

```
bss: load = block(0x40)
```

allocates .bss so that the entire section is contained in a single 64K-word page or begins on a page.

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

7.8.2.4 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 = 0800h,   length = 0240h
   PAGE 1 :  DATA    : origin = 0A00h,   length = 02200h
   PAGE 1 :  OVR_MEM : origin = 02D00,   length = 01000h
   PAGE 2 :  DATA    : origin = 0A00h,   length = 02200h
   PAGE 2 :  OVR_MEM : origin = 02D00,   length = 01000h
}
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. The linker combines input sections by concatenating them in the order in which they are specified. The size of an output section is the sum of the sizes of the input sections that it comprises.

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

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

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

In Example 7–5, the linker takes all the .text sections from the input files and combines them into the .text output section. The linker concatenates the .text input sections in the order in which it encounters them in the input files. The linker performs similar operations with the .data and .bss sections. You can use this type of specification for any output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
  .text :              /* Build .text output section        */
  {
    f1.obj(.text)      /* Link .text section from f1.obj    */
    f2.obj(sec1)       /* Link sec1 section from f2.obj     */
    f3.obj             /* Link ALL sections from f3.obj     */
    f4.obj(.text,sec2) /* Link .text and sec2 from f4.obj   */
   }
}
```

It is not necessary for the input sections that make up an output section to have the same name. It is also not necessary for the input sections to have the same name as the output section of which they become part. If a file is listed with no sections, *all* of its sections are included in the output section. If any additional input sections have the same name as an output section but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the linker found more .text sections in the preceding example and these .text sections *were not* specified anywhere in the SECTIONS directive, the linker would concatenate these extra sections after f4.obj (sec2).

The specifications in Example 7–5 are actually a shorthand method for the following:

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

The specification *(.text) means *the unallocated .text sections from all the input files.* This format is useful when:

- You want the output section to contain all input sections that have a specified name, but the output section's name is different than the input sections' names.
- You want the linker to allocate the input sections before it processes additional input sections or commands within the braces.

The following example illustrates the two cases described in the preceding list:

```
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. All unallocated sections are included. For example, if one of the .text input sections was already included in another output section when the linker encounters *(.text), the linker can not include that first .text input section in the second output section.

7.9 Specifying a Section's Run-Time Address

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

The linker provides a simple way to accomplish this. You can use the SECTIONS directive to direct the linker to allocate a section twice: once to set its load address and again to set its run address. For example:

```
.fir: load = SLOW_MEM, run = FAST_MEM
```

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

See to section 2.4.1, *Run-Time Relocation*, on page 2-16 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, *Overlaying Sections With the UNION Statement*, on page 7-45.)

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 encountered, 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 = ROM, align = 32, run = RAM /* align load only */
.data: load = (ROM align 32),  run = RAM /* align load only */
.data: run  = RAM, align 32  /* align 32 in RAM for run */
       load = align 16       /* align 16 anywhere for load */
```

7.9.2 Uninitialized Sections

Uninitialized sections (such as .bss) are not loaded, so their only significant address is the run address. The linker allocates uninitialized sections only once; if you specify both run and load addresses, the linker warns you and ignores the load address. If you specify only one address, the linker treats it as a run address, regardless of whether you call it load or run. This example specifies load and run addresses for an uninitialized section:

```
.bss: load = 0x1000, run = RAM
```

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

```
.bss: load = RAM
.bss: run = RAM
.bss: > RAM
```

7.9.3 Referring to the Load Address by Using the .label Directive

Normally any reference to a symbol in a section refers to its run-time address. However, it may be necessary at run time to refer to a load-time address. Specifically, the code that copies a section from its load address to its run address must have access to the load address. The .label directive defines a special symbol that refers to the section's load address. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address. For more information on the .label directive, see page 4-49.

Example 7–6 shows the use of the .label directive. Figure 7–4 illustrates the run-time execution of Example 7–6.

Example 7–6. Copying a Section From ROM to RAM

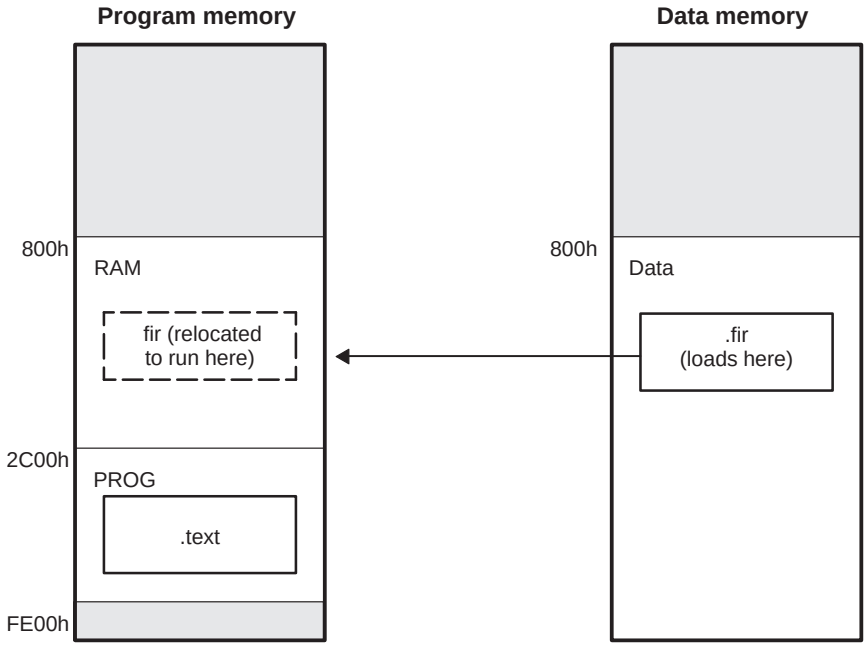
(a) Assembly language file

```
;----------------------------------------------------------
;  define a section to be copied from DATA to PROGRAM
;----------------------------------------------------------
    .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 DATA into PROGRAM

;----------------------------------------------------------
    .text

   MOV   XAR6, fir_src
   MOV   XAR7, #fir
   RPT  #(fir_end – fir_src – 1)

||  PWRITE *XAR7, *XAR6++

;----------------------------------------------------------
;  jump to section, now in RAM
;----------------------------------------------------------
     B  fir
```

(b) Linker command file

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

MEMORY
{
   PAGE 0 :  RAM   :  origin = 0800h,   length = 02400h
   PAGE 0 :  PROG  :  origin = 02C00h,  length = 0D200h
   PAGE 1 :  DATA  :  origin = 0800h,   length = 0F800h
}

SECTIONS
{
   .text: load = PROG PAGE 0
   .fir:  load = DATA PAGE 1, run RAM PAGE 0
}
```

Figure 7–4. Run-Time Execution of Example 7–6



7.10 Using UNION and GROUP Statements

Two SECTIONS statements allow you to conserve memory: GROUP and UNION. Unioning sections causes the linker to allocate them to the same run address. Grouping sections causes the linker to allocate them contiguously in memory. Section names can refer to 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 7–7, the .bss sections from file1.obj and file2.obj are allocated at the same address in RAM. In the memory map, the union occupies as much space as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

Example 7–7. The UNION Statement

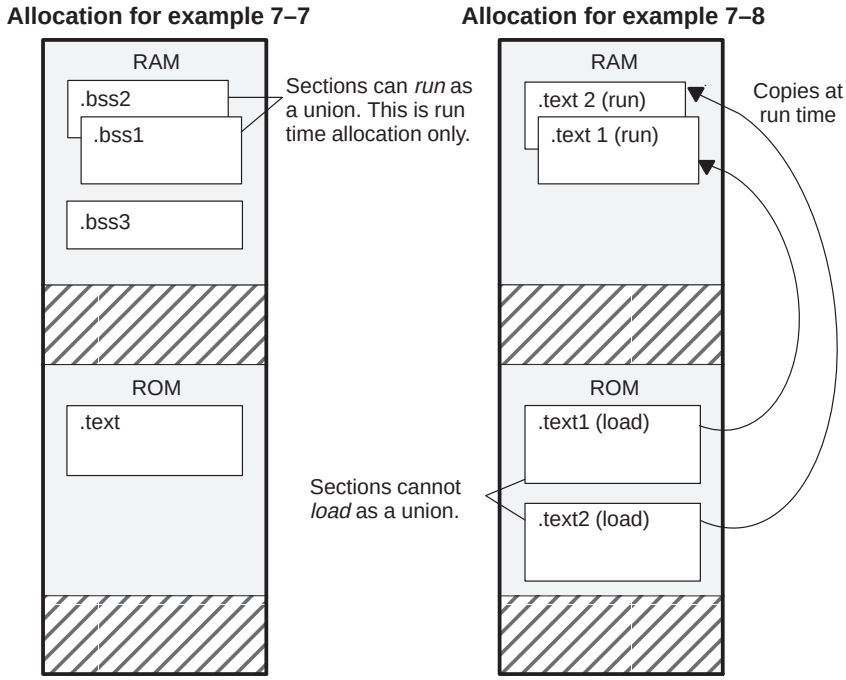
```
SECTIONS
{
   .text: load = ROM
   UNION: run = RAM
   {
      .bss1: { file1.obj(.bss) }
      .bss2: { file2.obj(.bss) }
   }
      .bss3: run = RAM { globals.obj(.bss) }
```

Allocation of a section as part of a union affects only its *run address.* Under no circumstances can sections be overlaid for loading. If an initialized section is a union member (an initialized section, such as .text, has raw data), its load allocation *must* be separately specified, as shown in Example 7–8.

Example 7–8. Separate Load Addresses for UNION Sections

```
   UNION: run = RAM
   {
      .text1: load = ROM, {  file1.obj(.text) }
      .text2: load = ROM, { file2.obj(.text) }
   }
```

Figure 7–5. Memory Allocation Shown in Example 7–7 and Example 7–8



Since the .text sections contain data, they cannot *load* as a union, although they can be *run* as a union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a UNION, the linker issues a warning and allocates load space anywhere it can in configured memory.

Uninitialized sections are not loaded and do not require load addresses.

The UNION statement applies only to allocation of run addresses, so it is meaningless to specify a load address for the union itself. For purposes of allocation, the union is treated as an uninitialized section: any one allocation specified is considered a run address, and if both run and load addresses are specified, the linker issues a warning and ignores the load address.

7.10.2 Grouping Output Sections Together

The SECTIONS directive's GROUP option forces several output sections to be allocated contiguously. For example, assume that a section named term_rec contains a termination record for a table in the .data section. You can force the linker to allocate .data and term_rec together:

Example 7–9. Allocate Sections Together

```
SECTIONS
{
   .text           /* Normal output section             */
   .bss            /* Normal output section             */
   GROUP 1000h :   /* Specify a group of sections       */
   {
      .data        /* First section in the group        */
      term_rec     /* Allocated immediately after .data */
   }
}
```

You can use binding, alignment, or named memory to allocate a GROUP in the same manner as a single output section. In the preceding example, the GROUP is bound to address 1000h. This means that .data is allocated at 1000h, and term_rec follows it in memory.

Note: You Cannot Specify Addresses for Sections Within a GROUP

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

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 linker to load various output sections into each of these banks or into banks that are not mapped at load time.

The linker supports this feature by providing *overlay pages*. Each page represents an address range that must be configured separately with the MEMORY directive. You then use the SECTIONS directive to specify the sections to be mapped into various pages.

Note: Overlay Section and Overlay Page Are Not the Same

The UNION capability and the *overlay page* capability (see section 7.10.1, *Overlaying Sections With the UNION Statement*, on page 7-45) 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 linker, each overlay page represents a completely separate memory space comprising the full range of addressable locations. In this way, you can link two or more sections at the same (or overlapping) addresses if they are on different pages.

Pages are numbered sequentially, beginning with 0. If you do not use the PAGE option, the linker allocates initialized sections into PAGE 0 (program memory) and uninitialized sections into PAGE 1 (data memory).

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 2BFF for PAGE 2. Although only one bank can be selected at a time, you can initialize each bank with different data. Example 7–10 shows how you use the MEMORY directive to obtain this configuration:

Example 7–10. Memory Directive With Overlay Pages

```
MEMORY
{
   PAGE 0   : RAM     :origin = 0800h,   length = 0240h
            : PROG    :origin = 02C00h,  length = 0D200h
   PAGE 1   : OVR_MEM :origin = 0A00h,   length = 02200h
            : DATA    :origin = 02C00h,  length = 0D400h
   PAGE 2   : OVR_MEM :origin = 0A00h,   length = 02200h
}
```

Example 7–10 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 7–10. 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 7–11 shows how to use the SECTIONS directive overlays to accomplish these objectives.

Example 7–11. SECTIONS Directive Definition for Overlays in Example 7–10

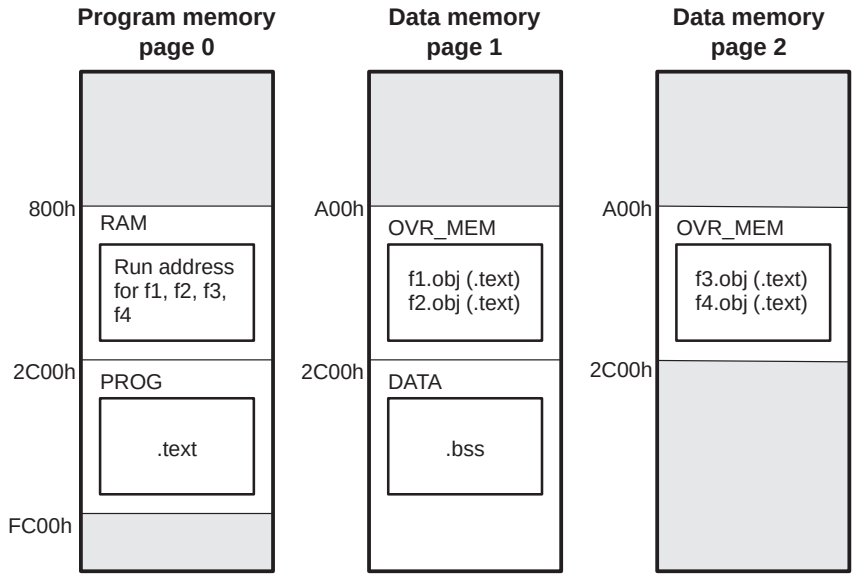
```
SECTIONS
{
   UNION :  run = RAM
   {
      S1 :  load = OVR_MEM PAGE 1
      {
         s1_load = 0A00h;
         s1_start = .;
         f1.obj (.text)
         f2.obj (.text)
         s1_length = . – s1_start;
      }
      S2 :  load = OVR_MEM PAGE 2
      {
         s2_load = 0A00h;
         s2_start = .;
         f3.obj (.text)
         f4.obj (.text)
         s2_length = . – s2_start;
      }
   }

   .text: load = PROG PAGE 0
   .data: load = PROG PAGE 0
   .bss : load = DATA PAGE 1
}
```

The four modules are f1, f2, f3, and f4. Modules f1 and f2 are combined into output section S1, and f3 and f4 are combined into output section S2. The PAGE specifications for S1 and S2 tell the linker to link these sections into the corresponding pages. As a result, they are both linked to load address A00h, but in different memory spaces. When the program is loaded, a loader can configure hardware so that each section is loaded into the appropriate memory bank.

7.11.4 Memory Allocation for Overlaid Pages

Figure 7–6 shows overlay pages defined by the MEMORY directive in Example 7–10 and the SECTIONS directive in Example 7–11.

Figure 7–6. Overlay Pages Defined in Example 7–10 and Example 7–11



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. 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 within it are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT section had actually been loaded. These symbols can be referenced by other input sections.
  - Undefined external symbols found within it cause specified archive libraries to be searched.
  - Its 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 0x00002000. 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 TMS320C27x C/C++ compiler has this attribute under the run-time initialization model.

- A NOLOAD section differs from a normal output section in one respect: its contents, relocation information, and line-number information are not placed in the output module. The linker allocates space for the section, and it appears in the memory-map listing.

7.13 Default Allocation

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections that you choose *not* to specify must still be handled by the linker. The linker uses default algorithms to build and allocate sections within the specifications you supply.

If you do not use the MEMORY and SECTIONS directives, the linker allocates output sections as though the definitions in Example 7–12 were specified.

Example 7–12. Default Allocation for TMS320C27x 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.

7.13.1 Output Section Formation

If you use a SECTIONS directive, the linker performs *no part* of the default allocation. Allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described in section 7.13.2.

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, *The SECTIONS Directive*, on page 7-31 for examples of how to define an output section's content.)

If an output section is formed by combining input sections not specified by a SECTIONS directive, the linker combines all such input sections that have the same name into an output section with that name. For example, suppose the files f1.obj and f2.obj both contain named sections called Vectors and that the SECTIONS directive does not define an output section for them. The linker combines the two Vectors sections from the input files into a single output section named Vectors, allocates it into memory, and includes it in the output file.

By default, the linker does not display a message when it creates an output section that is not defined in the SECTIONS directive. You can use the –w linker option (see section 7.4.18, *Display a Message When an Undefined Output Section Is Created (–w Option)*, on page 7-19) to cause the linker to display a message when it creates a new output section.

After the linker determines the composition of all output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured, if there is no MEMORY directive, the linker uses the default configuration as shown in Example 7–12. (See section 7.7, *The MEMORY Directive*, on page 7-26, for more information on configuring memory.)

7.13.2 Default Allocation Algorithm

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. The algorithm comprises these steps:

1) Each output section for which you have supplied a specific binding address is placed in memory at that address.

2) Each output section that is included in a specific named memory range or that has memory attribute restrictions is allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.

3) Any remaining sections are allocated in the order in which they are defined. Sections not defined in a SECTIONS directive are allocated in the order in which they are encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

7.14 Assigning Symbols at Link Time

Linker assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to initialize a variable or pointer to an allocation-dependent value.

7.14.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the C language:

symbol	=	*expression;*	assigns the value of expression to symbol
symbol	+=	*expression;*	adds the value of expression to symbol
symbol	–=	*expression;*	subtracts the value of expression from symbol
symbol	*=	*expression;*	multiplies symbol by expression
symbol	/=	*expression;*	divides symbol by expression

The symbol should be defined externally. If it is not, the linker defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in section 7.14.3, *Assignment Expressions.* Assignment statements *must* terminate with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Therefore, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol cur_tab as the address of the current table. The cur_tab symbol must point to either Table1 or Table2. You could accomplish this in the assembly code, but you would need to reassemble the program to change tables. Instead, you can use a linker assignment statement to assign cur_tab at link time:

```
prog.obj            /* Input file                          */
cur_tab = Table1;   /* Assign cur_tab to one of the tables */
```

7.14.2 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 linker's . symbol is analogous to the assembler's $ symbol. The . symbol can be used only in assignment statements within a SECTIONS directive because . is meaningful only during allocation and SECTIONS controls the allocation process. (See section 7.8, *The SECTIONS Directive*, on page 7-31.)

The . symbol refers to the current run address, not the current load address, of the section.

Suppose a program needs to know the address of the beginning of the .data section. By using the .global directive (see page 4-44), you can create an external undefined variable called Dstart. Then, assign the value of . to Dstart:

```
SECTIONS
{
   .text:    {}
   .data:    { Dstart = .; }
   .bss :    {}
}
```

This defines Dstart to be the first linked address of the .data section. (Dstart is assigned *before* .data is allocated.) The linker relocates all references to Dstart.

A special type of assignment assigns a value to the . symbol. This adjusts the SPC within an output section and creates a hole between two input sections. Any value assigned to . to create a hole is relative to the beginning of the section, not to the address actually represented by the . symbol. Holes and assignments to . are described in section 7.15, *Creating and Filling Holes*, on page 7-59.

7.14.3 Assignment Expressions

These rules apply to linker 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 linker in the same way they are identified by the assembler. That is, numbers are recognized as decimal unless they have a suffix (H or h for hexadecimal and Q or q for octal). C language prefixes are also recognized (0 for octal and 0x for hex). Hexadecimal constants must begin with a digit. No binary constants are allowed.
- Symbols within an expression have only the value of the symbol's *address*. No type checking is performed.
- Linker expressions can be absolute or relocatable. If an expression contains *any* relocatable symbols (and zero or more constants or absolute symbols), it is relocatable. Otherwise, the expression is absolute. If a symbol is assigned the value of a relocatable expression, it is relocatable; if it is assigned the value of an absolute expression, it is absolute.

The linker supports the C language operators that are listed in Table 7–2 in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in Table 7–2, the linker also has an align operator that allows a symbol to be aligned on an n-byte boundary within an output section (n is a power of 2). For example, the expression

```
. = align(16);
```

aligns the SPC within the current section on the next 16-byte boundary. Because the align operator is a function of the current SPC, it can be used only in the same context as .; that is, within a SECTIONS directive.

The statement `. = align(16)` will cause the output section to align on a 16-word boundary. Any assignment to the . symbol with the special align operator will align the affected output section the same way. If multiple align statements are used, the output section is aligned on the largest alignment specified.

Table 7–2. Groups of Operators Used in Expressions on Order of Precedence

Group 1 (Highest Precedence)		**Group 6**	
!	Logical NOT	&	Bitwise AND
~	Bitwise NOT		
–	Negation		
Group 2		**Group 7**	
*	Multiplication	|	Bitwise OR
/	Division		
%	Modulus		
Group 3		**Group 8**	
+	Addition	&&	Logical AND
–	Subtraction		
Group 4		**Group 9**	
>>	Arithmetic right shift	||	Logical OR
<<	Arithmetic left shift		
Group 5		**Group 10 (Lowest Precedence)**	
==	Equal to	=	Assignment
! =	Not equal to	+ =	A + = B → A = A + B
>	Greater than	– =	A – = B → A = A – B
<	Less than	* =	A * = B → A = A * B
< =	Less than or equal to	/ =	A / = B → A = A / B
> =	Greater than or equal to		

7.14.4 Symbols Defined by the Linker

The linker automatically defines several symbols, which vary depending on which sections are used in your assembly source. A program can use these symbols at run time to determine where a section is linked. Since these symbols are external, they appear in the linker map. Each symbol can be accessed in any assembly language module if it is declared with a .global directive (see page 4-44). You must have used the corresponding section in a source module for the symbol to be created. Values are assigned to these symbols as follows:

.text is assigned the first address of the .text output section. (It marks the *beginning* of executable code.)

etext is assigned the first address following the .text output section. (It marks the *end* of executable code.)

.data is assigned the first address of the .data output section. (It marks the *beginning* of initialized data tables.)

edata is assigned the first address following the .data output section. (It marks the *end* of initialized data tables.)

.bss is assigned the first address of the .bss output section. (It marks the *beginning* of uninitialized data.)

end is assigned the first address following the .bss output section. (It marks the *end* of uninitialized data.)

7.14.4.1 Symbols Defined Only for C Support (–c or –cr Option)

__STACK_SIZE is assigned the size of the .stack section.

__SYSMEM_SIZE is assigned the size of the .sysmem section.

7.15 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called *holes*. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles holes and how you can fill holes (and uninitialized sections) with values.

7.15.1 Initialized and Uninitialized Sections

An output section contains either:

- Raw data for the *entire* section
- *No* raw data

A section that has raw data is referred to as *initialized*. This means that the object file contains the actual memory-image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections *always* have raw data if anything was assembled into them. Named sections defined with the .sect assembler directive also have raw data.

By default, the .bss section (see page 4-26) and sections defined with the .usect directive (see page 4-75) 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 linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must supply raw data for the hole.*

Holes can be created only *within* output sections. Space can exist *between* output sections, but such space is not a hole. To fill the space between output sections, see section 7.7.2, *MEMORY Directive Syntax*, on page 7-26.

To create a hole in an output section, you must use a special type of linker assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by .) by adding to it, assigning a greater value to it, or aligning it on an address boundary. The operators, expressions, and syntaxes of assignment statements are described in section 7.14, *Assigning Symbols at Link Time*, on page 7-55.

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

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

The output section outsect is built as follows:

1) The .text section from file1.obj is linked in.
2) The linker creates a 256-word hole.
3) The .text section from file2.obj is linked in after the hole.
4) The linker creates another hole by aligning the SPC on a 16-word boundary.
5) Finally, the .text section from file3.obj is linked in.

All values assigned to the . symbol within a section refer to the *relative address within the section.* The linker handles assignments to the . symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement . = align(16) in the example. This statement aligns outsect on a 16-word boundary. Any assignment to the . symbol with the special align operator will align the affected output section in the same way. If multiple align statements are used, the output section is aligned on the largest alignment specified.

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:   {  .+= 100h; }     /* Hole at the beginning */
.data:   {
            *(.data)
            . += 100h; }    /* Hole at the end       */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* The following example illustrates this method:

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      file1.obj(.bss)          /* This becomes a hole */
   }
}
```

Because the .text section has raw data, all of outsect must also contain raw data. Therefore, the uninitialized .bss section becomes a hole.

Uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

7.15.3 Filling Holes

When a hole exists in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 16-bit fill value that is replicated through memory until it fills the hole. The linker determines the fill value as follows:

1) If the hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = sign and a 16-bit constant:

```
SECTIONS
{
      outsect:
   {
      file1.obj(.text)
      file2.obj(.bss) = 00FFh   /* Fill this hole  */
   }                            /* with 0FFh */
}
```

2) You can also specify a fill value for all the holes in an output section by supplying the fill value after the section definition:

```
SECTIONS
{
   outsect:fill = 0FF00h
                          /* Fills holes with 0FF00h */
   {
      . += 10h;           /* This creates a hole         */
      file1.obj(.text)
      file1.obj(.bss)     /* This creates another hole   */
   }
}
```

3) If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with the –f option (see section 7.4.5, *Set Default Fill Value (–f* fill_value *Option)*, on page 7-11). For example, suppose the command file link.cmd contains the following SECTIONS directive:

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

Now invoke the linker with the –f option:

```
lnk2000 –f 0FFFFh link.cmd
```

This fills the hole with 0FFFFh.

4) If you do not invoke the linker with the –f option or otherwise specify a fill value, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

7.15.4 Explicit Initialization of Uninitialized Sections

You can force the linker to initialize an uninitialized section by specifying an explicit fill value for it in the SECTIONS directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
   .bss: fill = 1234h    /* Fills .bss with 1234h */
}
```

Note: Filling Sections

Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large sections or holes.

7.16 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as *partial linking* or *incremental linking*. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program.

Follow these guidelines for producing a file that you will relink:

- ❏ The intermediate files produced by the linker *must* have relocation information. Use the –r option on all links except the last one. (See section 7.4.1, *Relocation Capabilities (–a and –r Options)*, on page 7-8.)
- ❏ Intermediate files *must* have symbolic information. By default, the linker retains symbolic information in its output. Do not use the –s option if you plan to relink a file, because –s strips symbolic information from the output module. (See section 7.4.15, *Strip Symbolic Information (–s Option)*, on page 7-18.)
- ❏ 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 –h option (see section 7.4.7, *Make All Global Symbols Static (–h Option)*, on page 7-11).
- ❏ If you are linking C code, do not use –c or –cr until the final link step. Every time you invoke the linker with the –c or –cr option, the linker attempts to create an entry point. (See section 7.4.3, *C Language Options (–c and –cr Options)*, on page 7-10.)
- ❏ Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated; that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it.

The following example shows how you can use partial linking:

Step 1: Link the file file1.com; use the –r option to retain relocation information in the output file tempout1.out.

```
lnk2000 –r –o tempout1 file1.com
```

file1.com contains:

```
SECTIONS
{
   ss1:   {
          f1.obj
          f2.obj
           .
           .
           .
          fn.obj
          }
}
```

Step 2: Link the file file2.com; use the –r option to retain relocation information in the output file tempout2.out.

```
lnk2000 –r –o tempout2 file2.com
```

file2.com contains:

```
SECTIONS
{
   ss2:   {
          g1.obj
          g2.obj
           .
           .
           .
          gn.obj
          }
}
```

Step 3: Link tempout1.out and tempout2.out.

```
lnk2000 –m final.map –o final.out tempout1.out tempout2.out
```

7.17 Linking 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:

```
lnk2000 -c –o prog.out prog1.obj prog2.obj ... rts.lib
```

The –c option tells the linker to use special conventions that are defined by the C environment. The archive library rts.lib contains C run-time-support functions.

For more information about C, including the run-time environment and run-time-support functions, see the *TMS320C27x Optimizing C/C++ Compiler User's Guide.*

7.17.1 Run-Time Initialization

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

1) Sets up the system stack

2) Processes the run-time initialization table and autoinitializes global variables (when the linker is invoked with the –c option)

3) Calls _main

The run-time-support object library, rts.lib, contains boot.obj. You can:

- ❏ Include rts.lib as an input file (the linker automatically extracts boot.obj when you use the –c or –cr option)
- ❏ Use the archiver to extract boot.obj from the library and then link the module in directly

7.17.2 Object Libraries and Run-Time Support

The *TMS320C27x Optimizing C/C++ Compiler User's Guide* describes additional run-time-support functions that are included in rts.src. If your program uses any of these functions, you must link *rts.lib* with your object files.

You can also create your own object libraries and link them. The linker includes and links only those library members that resolve undefined references.

7.17.3 Setting the Size of the Stack and Heap Sections

C uses two uninitialized sections called .sysmem and .stack for the memory pool used by the malloc( ) functions and the run-time stacks, respectively. You can set the size of these by using the –heap or –stack option and specifying the size of the section as a 4-byte constant immediately after the option. The default size for both, if the options are not used, is 1K words.

See section 7.4.8, *Define Heap Size (–heap* Size *Option)*, on page 7-12 and section 7.4.16, *Define Stack Size (–stack* Size *Option)*, on page 7-18 for more information on setting stack sizes.

Note: Linking the .stack Section

The .stack section has to 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.

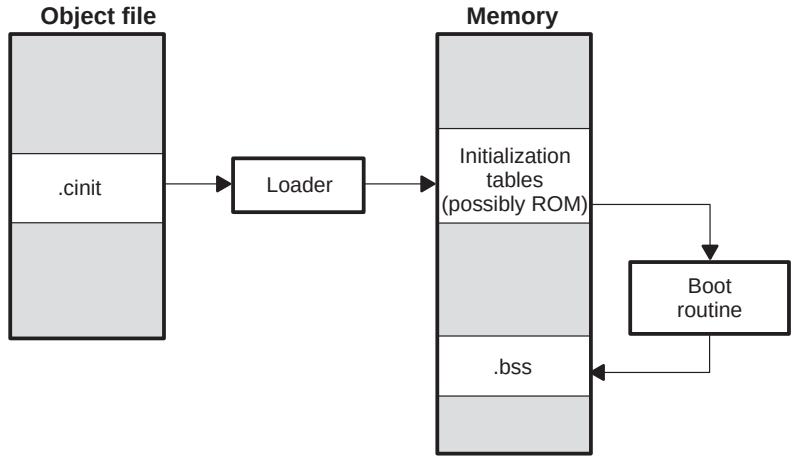
7.17.4 Autoinitialization of Variables at Run Time

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

Using this method, the .cinit section is loaded into memory along with all other initialized sections. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 7–7 illustrates autoinitialization at run time.

Figure 7–7. Autoinitialization at Run Time



7.17.5 Autoinitialization of Variables at Load Time

Autoinitialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the –cr option.

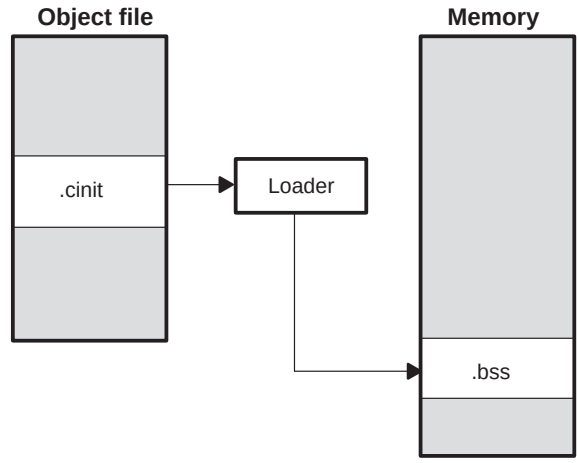
When you use the –cr linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to –1 (normally, cinit points to the beginning of the initialization tables). This setting indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use autoinitialization at load time:

- Detect the presence of the .cinit section in the object file
- Determine that STYP_COPY is set in the .cinit section header so that it knows not to copy the .cinit section into memory
- Understand the format of the initialization tables

Figure 7–8 illustrates the autoinitialization of variables at load time.

Figure 7–8. Autoinitialization at Load Time



7.17.6 The –c and –cr Linker Options

The following list outlines what happens when you invoke the linker with the –c or –cr option.

- The symbol _c_int00 is defined as the program entry point. The _c_int00 symbol is the start of the C boot routine in boot.obj; referencing _c_int00 ensures that boot.obj is automatically linked in from the run-time-support library rts.lib.
- The .cinit output section is padded with a termination record to designate to the boot routine (if you autoinitialize at run time) or the loader (if you autoinitialize at load time) when to stop reading the initialization tables.
- When you autoinitialize at run time (–c option), the linker defines cinit as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.
- When you autoinitialize at load time (–cr option):
  - The linker sets cinit to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag (0010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.

Note: Boot Loader

A loader is not included as part of the C/C++ compiler tools. Use the TMS320C27x Code Composer Studio as a loader.

7.18 Linker Examples

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 configuration:

Memory Type	Address Range	Contents
Program	`0xf0000 to 0x3fffbf`	ROM
	`0x3fffc0 to 0x3fffff`	Interrupt vector table
Data	`0x000040 to 0x0001ff`	Stack
	`0x000200 to 0x0007ff`	RAM_1
	`0x3ed000 to 0x3effff`	RAM_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 RAM_2.
- Tables of coefficients in the .data sections of demo.obj, tables.obj, and fft.obj are linked into RAM_1. A hole is created with a length of 100 and a fill value of 07A1Ch.
- The xy section from demo.obj, which contains buffers and variables, is linked by default into page one of the block STACK, since it is not explicitly linked.

Example 7–13 shows the linker command file for this example. Example 7–14 shows the map file.

Example 7–13. Linker Command File, demo.cmd

```
/*************************************************************/
/***                 Specify Linker Options                ***/
/*************************************************************/
-o demo.out               /* Name the output file             */
-m demo.map               /* Create an output map             */
/*************************************************************/
/***                  Specify the Input Files              ***/
/*************************************************************/
demo.obj
fft.obj
tables.obj
/*************************************************************/
/***            Specify the Memory Configuration           ***/
/*************************************************************/
MEMORY
{
       PAGE 0:  ROM      (R):   origin=3f0000h    length=0ffc0h
                VECTORS  (R):   origin=3fffc0h    length=0040h

       PAGE 1:  STACK    (RW):  origin=000040h    length=01c0h
                RAM_1    (RW):  origin=000200h    length=0600h
                RAM_2    (RW):  origin=3ed000h    length=3000h
}

/*************************************************************/
/***              Specify the Output Sections              ***/
/*************************************************************/
SECTIONS
{
       vectors    : { } > VECTORS page=0
       .text      : load = ROM, page = 0 /* link .text into ROM */
       .data      : fill = 07A1Ch, Load=RAM_1, page=1
       {
                  tables.obj(.data)             /*.data input */
                  fft.obj(.data)               /* .data input */
                  . += 100h;  /* create hole, fill with 07A1Ch */
       }
       var_defs   : { } > RAM_2 page=1      /* defs in RAM */
       .bss:      page=1, fill=0ffffh       /*.bss fill and link*/
}

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

Invoke the linker by entering the following command:

lnk2000 demo.cmd

This creates the map file shown in Example 7–14 and an output file called demo.out that can be run on a TMS320C27x device.

Example 7–14. Output Map File, demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: 0

MEMORY CONFIGURATION
             name      origin    length     attributes    fill
            --------   --------  ---------   ----------  --------
PAGE 0:     ROM        003f0000  0000ffc0     R
            VECTORS    003fffc0  00000040     R
PAGE 1:     STACK      00000040  000001c0     RW
            RAM_1      00000200  00000600     RW
            RAM_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)
```

Example 7–14. Output Map File, demo.map (Continued)

```
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
0000030c   ___edata__
003f0000   _main
003f0000   .text
003f0000   ___text__
003f000e   _func1
003f001a   etext
003f001a   ___etext__

[16 symbols]
```

Chapter 8

Absolute Lister Description

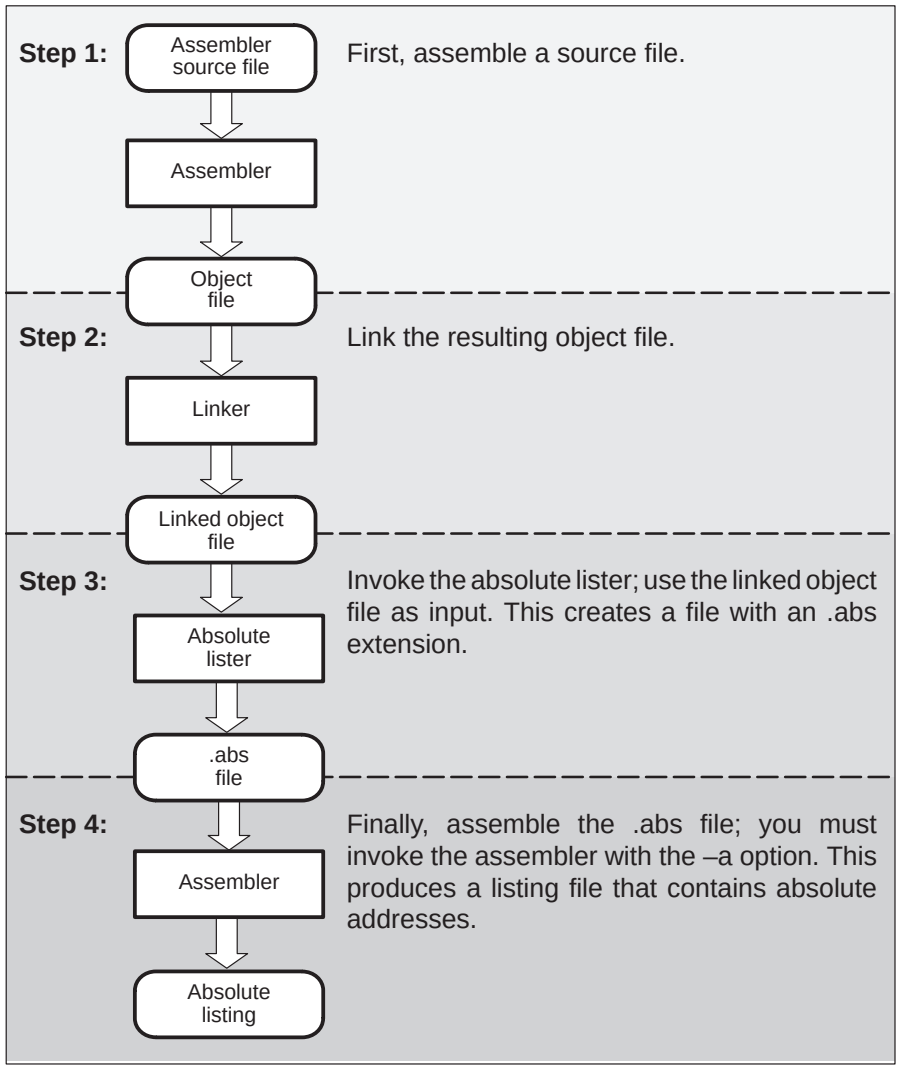
The TMS320C27x™ absolute lister is a debugging tool that accepts linked object files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code. Manually, this could be a tedious process requiring many operations; 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)

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

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

input file names the linked object file. If you do not supply an extension, the absolute lister assumes that the input file has the default extension .out. If you do not supply an input filename when you invoke the absolute lister, the absolute lister prompts you for one.

The absolute lister produces an output file for each file that was linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

Assemble these files with the –a assembler option as follows to create the absolute listing:

```
asm2000 –a filename.abs
```

The –e options affect both the interpretation of filenames on the command line and the names of the output files. They should always precede any filename on the command line.

The –e options are useful when the linked object file was created from C files compiled with the debugging option (–g compiler option). When the debugging option is set, the resulting linked object file contains the name of the source files used to build it. In this case, the absolute lister 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:

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

```
asm2000 module1
asm2000 module2
```

This command sequence creates two object files called module1.obj and module2.obj.

Step 2: Next, link module1.obj and module2.obj using the following linker command file, called bttest.cmd:

```
/************************************************/
/* File bttest.cmd -- COFF linker command file  */
/*       for linking TMS320C27x modules         */
/********************************** ************/
-o bttest.out           /* Name the output file */
-m bttest.map           /* Create an output map */

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

/************************************************/
/*       Specify the Memory Configurations      */
/************************************************/
MEMORY
{
   PAGE 0:   ROM:  origin=2000h   length=2000h
   PAGE 1:   RAM:  origin=8000h   length=8000h
}

/************************************************/
/*         Specify the Output Sections          */
/************************************************/
SECTIONS
{
   .data:  >RAM
   .text:  >ROM
   .bss:   >RAM
}
```

Invoke the linker:

`lnk2000 bttest.cmd`

This command creates an executable object file called bttest.out; use this new file as input for the absolute lister.

Step 3: Now, invoke the absolute lister:

```
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    ".pst",000008000h
            .list
            .text
            .copy       "module1.ism"
```

module2.bass:

```
            .molest
array       .setsym     000008000h
FDA         .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",000008066h
            .list
            .text
            .copy       "module2.asm"
```

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

- They contain .setsym directives, which equate values to global symbols. Both files contain global equates for the symbol *dflag*. The symbol dflag was defined in the file globals.def, which was included in module1.asm and module2.asm.
- They contain .setsect directives, which define the absolute addresses for sections.
- They contain .copy directives, which tell the assembler which assembly language source file to include.

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

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

```
asm2000  –a  module1.abs
asm2000  –a  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 Figure 8–2) and module2.lst (see Figure 8–3).

Figure 8–2. 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

 No Errors, No Warnings
```

Figure 8–3. 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

 No Errors, No Warnings
```

Chapter 9

Cross-Reference Lister Description

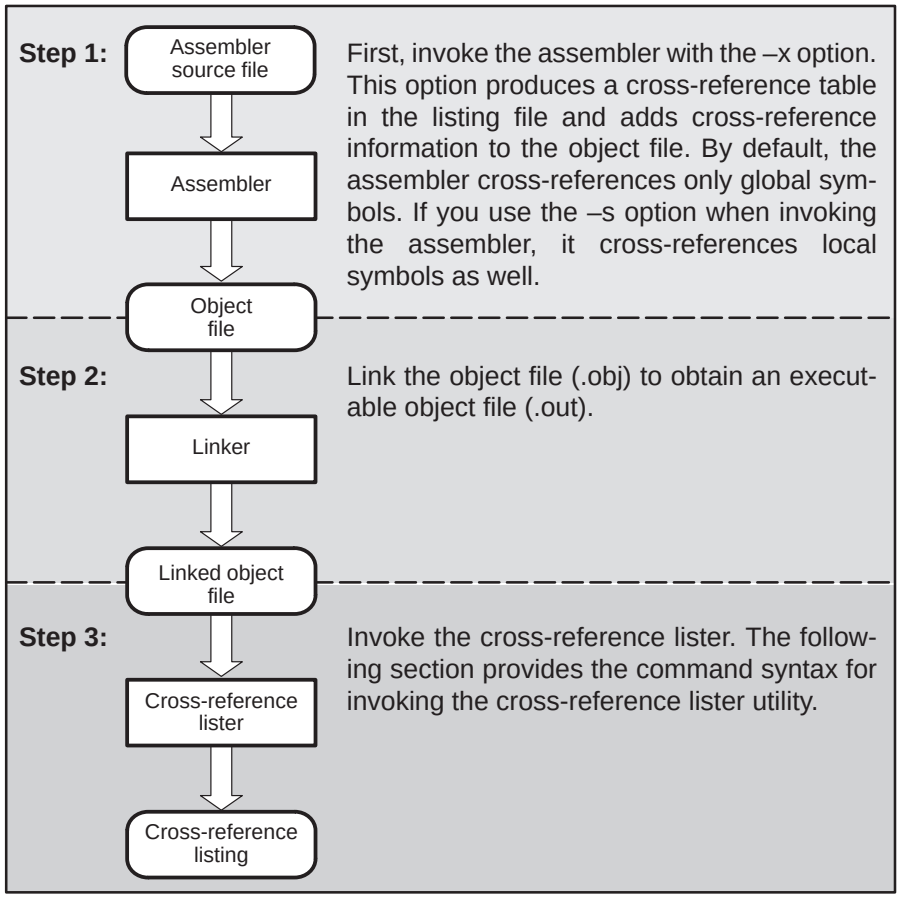
The TMS320C27x™ cross-reference lister is a debugging tool. This utility accepts linked object files as input and produces a cross-reference listing as output. This listing shows symbols, their definitions, and their references in the linked source files.

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. Cross-Reference Lister Development Flow



9.2 Invoking the Cross-Reference Lister

To use the cross-reference utility, the file must be assembled with the correct options and then linked into an executable file. Assemble the assembly language files with the –x option. This option create a cross-reference listing and adds cross-reference information to the object file. By default the assembler cross references only global symbols, but if the assembler is invoked with the –s option, local symbols are also added. Link the object files to obtain an executable file.

To invoke the cross-reference lister, use the following syntax:

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. Precede each option with a hyphen (–). The cross-reference lister options are as follows:

- **–l** (lowercase L) specifies the number of lines per page for the output file. The format of the –l option is –l*num*, where num is a decimal constant. For example, –l30 sets the number of lines per page in the output file to 30. The space between the option and the decimal constant is optional. The default is 60 lines per page.
- **–q** 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

```
================================================================================

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

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

Chapter 10

Hex-Conversion Utility Description

The TMS320C27x™ assembler and linker create object files that are in common object file format (COFF). COFF is a binary object file format that encourages modular programming and provides more powerful and flexible methods for managing code segments and target system memory.

Most EPROM programmers do not accept COFF object files as input. The hex-conversion utility converts a COFF object file into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer. The utility is also useful in other applications requiring hexadecimal conversion of a COFF object file (for example, debugger and loader applications).

The hex-conversion utility can produce these output file formats:

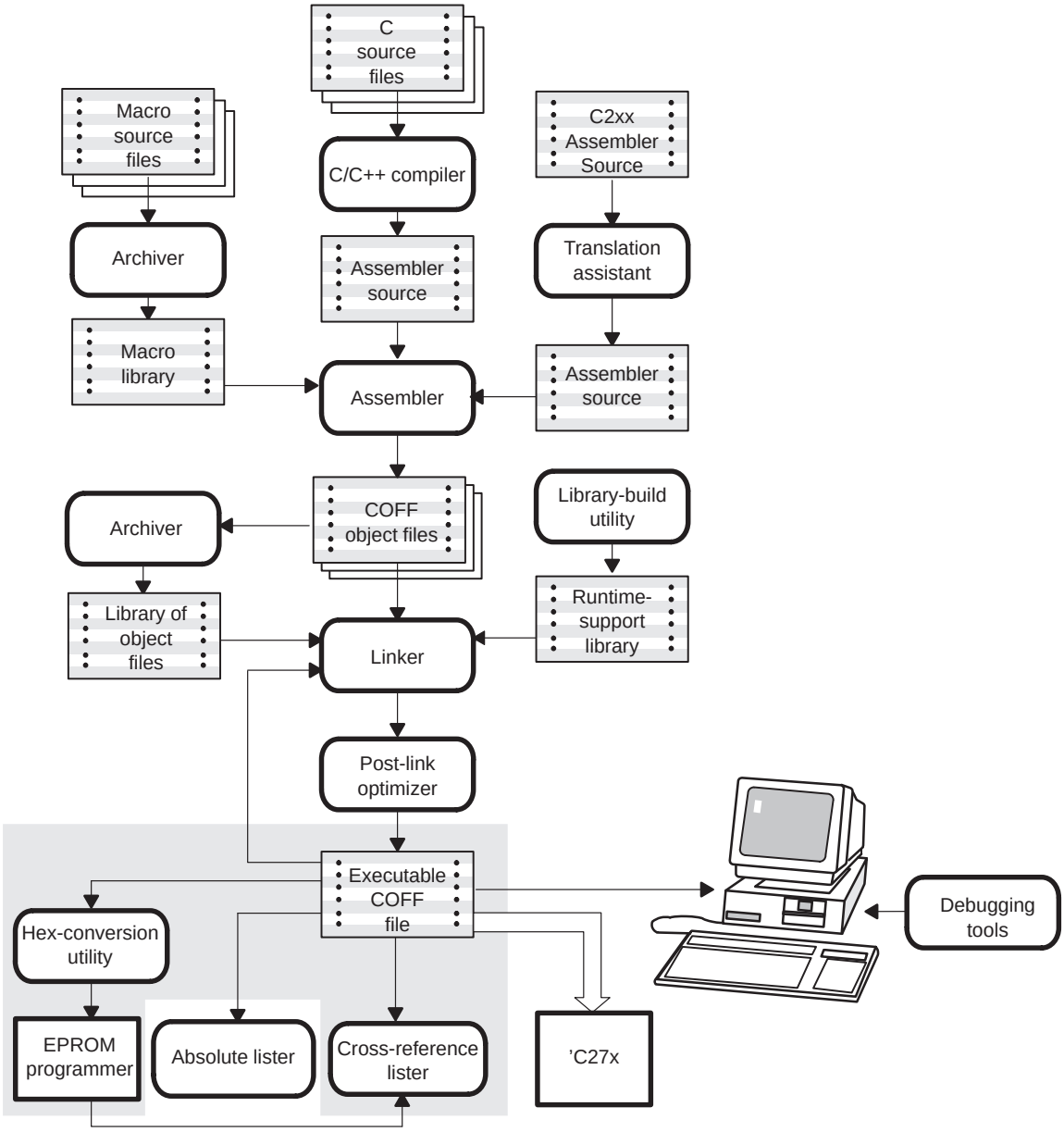
- ASCII-Hex, supporting 16-bit addresses
- Extended Tektronix (Tektronix)
- Intel MCS-86 (Intel)
- Motorola Exorciser (Motorola-S), supporting 16-bit addresses
- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses

Topic **Page**

10.1 The Hex-Conversion Utility's Role in the Software Development Flow

Figure 10–1 highlights the role of the hex-conversion utility in the software development process.

Figure 10–1. The Hex-Conversion Utility in the TMS320C27x Software Development Flow



10.2 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.

10.2.1 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.

- 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 is the –q (quiet) option, which must be used before any other options.

filename names a COFF object file or a command file (for more information, see section 10.2.2, *Invoking the Hex-Conversion Utility With a Command File*, on page 10-5).

Table 10–1. Basic Options

General Options	Option	Description	Page
Control the overall operation of the hex-conversion utility.	–map *filename*	Generate a map file	10-18
	–o *filename*	Specify an output filename	10-22
	–q	Run quietly (when used, it must appear *before* other options)	10-5
Image Options	**Option**	**Description**	**Page**
Create a continuous image of a range of target memory.	–fill *value*	Fill holes with *value*	10-25
	–image	Specify image mode	10-24
	–zero	Reset the address origin to zero in image mode	10-26
Memory Options	**Option**	**Description**	**Page**
Configure the memory widths for your output files.	–memwidth *value*	Define the system memory word width (default 32 bits)	10-8
	–romwidth *value*	Specify the ROM device width (default depends on format used)	10-9
Output Formats	**Option**	**Description**	**Page**
Specify the output format.	–a	Select ASCII-Hex	10-28
	–i	Select Intel	10-29
	–m	Select Motorola-S	10-30
	–t	Select TI-Tagged	10-31
	–x	Select Tektronix	10-32

10.2.2 Invoking the Hex-Conversion Utility With a Command File

A command file is useful when you plan to invoke the utility more than once with the same input files and options. It is also useful when you want to use the ROMS and SECTIONS hex-conversion utility directives to customize the conversion process.

Command files are ASCII files that contain one or more of the following:

- **Options and filenames.** These are specified in a command file in exactly the same manner as on the command line.
- **ROMS directive.** The ROMS directive defines the physical memory configuration of your system as a list of address-range parameters. (For more information, see section 10.4, *The ROMS Directive*, on page 10-14.)
- **SECTIONS directive.** The SECTIONS directive specifies which sections from the COFF object file are selected. (For more information, see section 7.8, *The SECTIONS Directive*, on page 7-31.)
- **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
```

The following example shows how to convert a file called appl.out into four hex files in Intel format. Each output file is one byte wide and 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 }
ROW2: origin=0x00004000 len=0x4000 romwidth=8
                files={ app1.u2 appl.u3 }
                }

              SECTIONS
{   .text, .data, .cinit, .sect1, .vectors, .const:
                }
```

10.3 Understanding Memory Widths

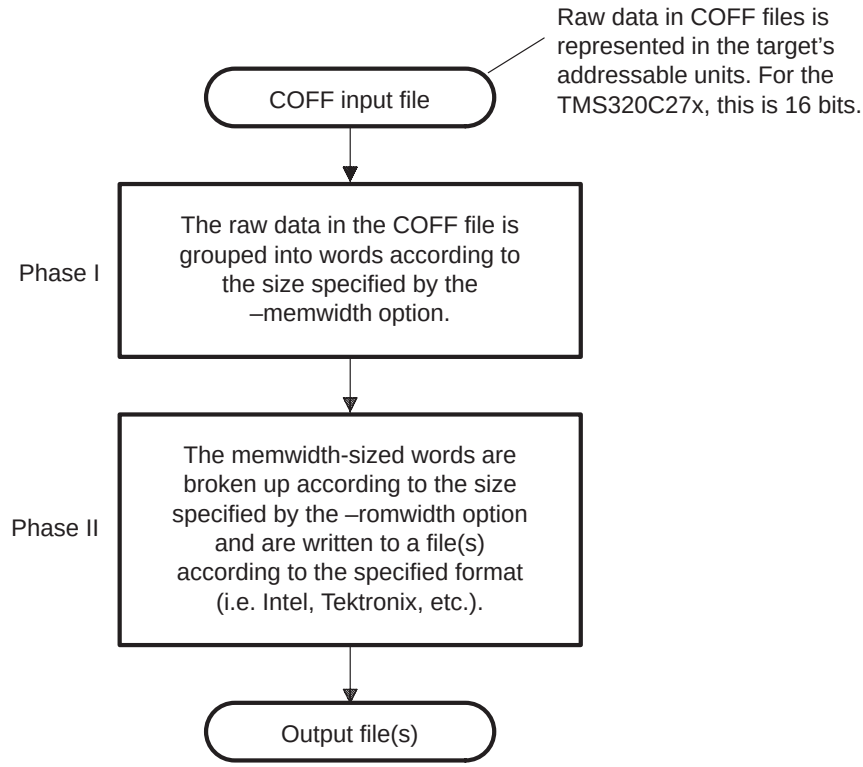
The hex-conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex-conversion utility, *you must understand how the utility treats word widths*. Three widths are important in the conversion process:

- Target width
- Memory width
- ROM width

The terms *target word*, *memory word*, and *ROM word* refer to a word of target, memory, and ROM width, respectively.

Figure 10–2 illustrates the two separate and distinct phases of the hex-conversion utility's process flow.

Figure 10–2. Hex-Conversion Utility Process Flow



10.3.1 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 TMS320C27x targets have a width of 32 bits.

10.3.2 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 narrower multiple consecutive memory words.

The hex-conversion utility defaults memory width to the target width (in this case, 32 bits).

You can change the memory width by:

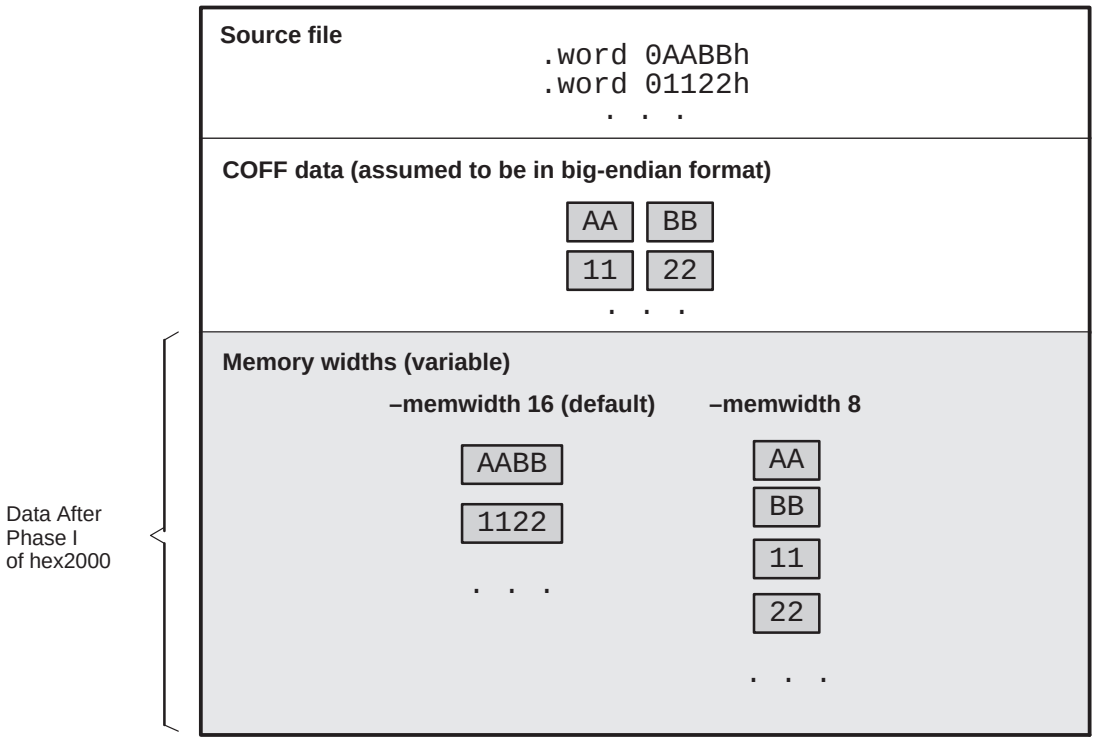
- Using the **–memwidth** option. This changes the memory-width value for the entire file.
- Setting the **memwidth** parameter of the ROMS directive. This changes the memory-width value for the address range specified in the ROMS directive and overrides the –memwidth option for that range. See section 10.4, *The ROMS Directive*, on page 10-14.

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 narrower consecutive memory words.

Figure 10–3 demonstrates how the memory width is related to COFF data.

Figure 10–3. COFF Data and Memory Widths



10.3.3 Partitioning Data Into Output Files

ROM width specifies the physical width (in bits) of each ROM device and corresponding output file (usually one byte or eight bits). The ROM width determines how the hex-conversion utility partitions the data into output files. After the COFF data is mapped to the memory words, the memory words are broken into one or more output files. The number of output files is determined by the following formulas:

- If memory width ≥ ROM width:

  number of files = memory width ÷ ROM width

- If memory width < ROM width:

  number of files = 1

For example, for a memory width of 16, you could specify a ROM-width value of 16 and get a single output file containing 16-bit words. Or you can use a ROM-width value of 8 to get two files, each containing 8 bits of each word.

The default ROM width that the hex-conversion utility uses depends on the output format:

- All hex formats except TI-Tagged are configured as lists of 8-bit bytes; the default ROM width for these formats is eight bits.
- TI-Tagged is a 16-bit format; the default ROM width for TI-Tagged is 16 bits.

Note: The TI-Tagged Format Is 16 Bits Wide

You cannot change the ROM width of the TI-Tagged format. The TI-Tagged format supports a 16-bit ROM width only.

You can change ROM width (except for TI-Tagged) by:

- Using the **–romwidth** option. This option changes the ROM-width value for the entire COFF file.
- Setting the **romwidth** parameter of the ROMS directive. This parameter changes the ROM width value for a specific ROM address range and overrides the –romwidth option for that range. See section 10.4, *The ROMS Directive*, on page 10-14.

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 10–4 illustrates how the COFF data, memory, and ROM widths are related to one another.

Memory width and ROM width are used only for grouping the COFF data; they do not represent values. Thus, the byte ordering of the COFF data is maintained throughout the conversion process. To refer to the partitions within a memory word, the bits of the memory word are always numbered from right to left as follows:

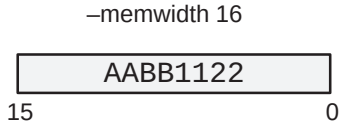
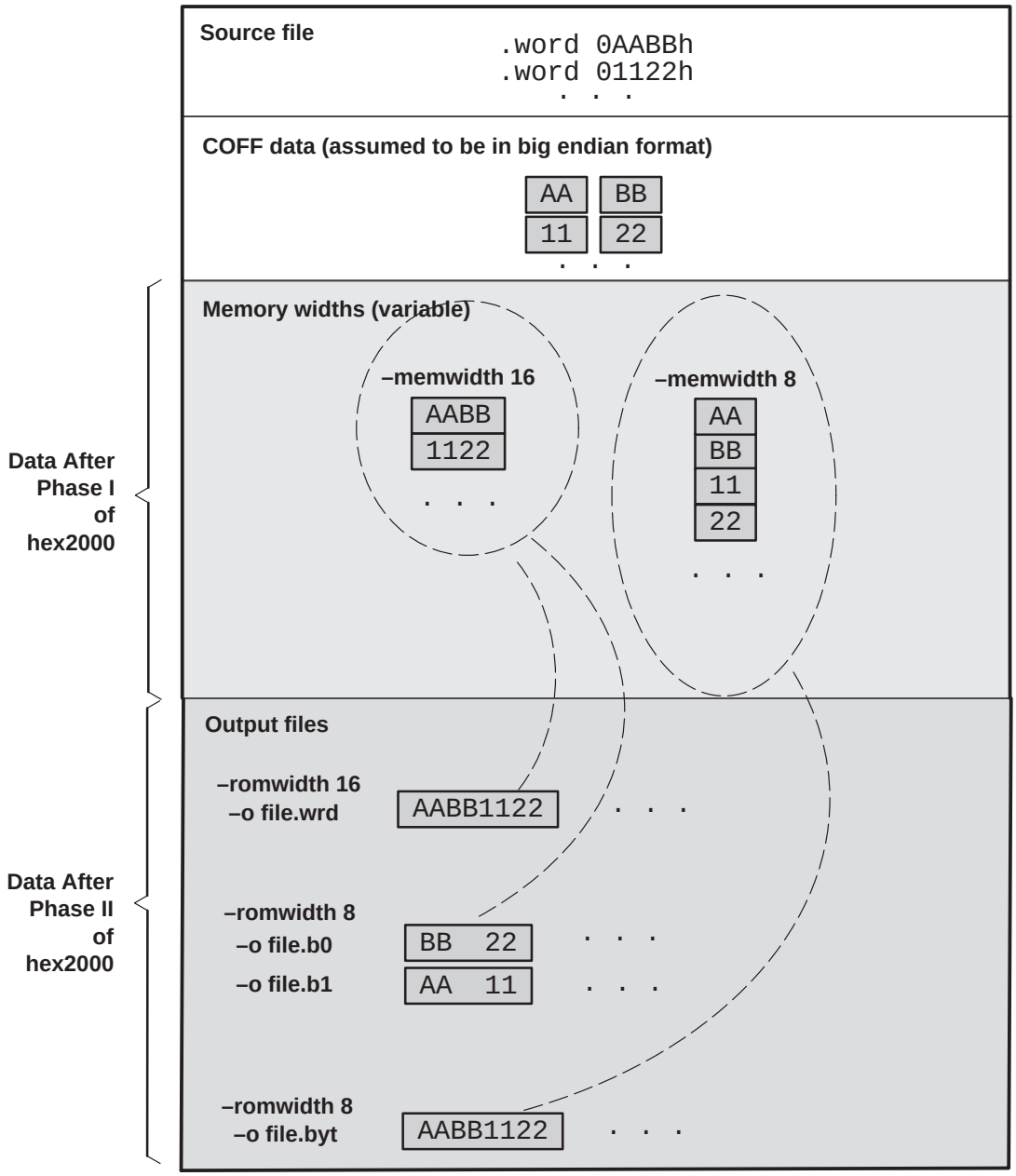


Figure 10–4. Data, Memory, and ROM Widths



10.3.4 Specifying Word Order for Output Words

When memory words are narrower than target words (memory width < 16), target words are split into multiple consecutive memory words. There are two ways to split a wide word into consecutive memory locations in the same hex conversion utility output file:

- **–order MS** specifies **big-endian** ordering, in which the most significant part of the wide word occupies the first of the consecutive locations
- **–order LS** specifies **little-endian** ordering, in which the the least significant part of the wide word occupies the first of the consecutive locations

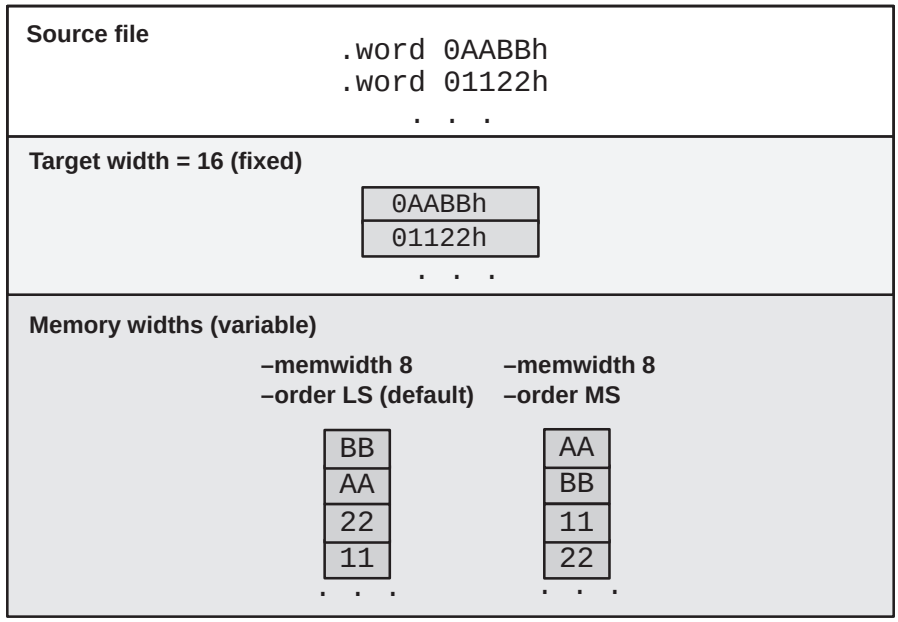
By default, the utility uses little-endian format because the TMS320C27x boot loaders expect the data in this order. Unless you are using your own boot loader program, avoid using –order MS.

Note: When the –order Option Applies

- This option applies only when you use a memory width with a value less than 16. Otherwise, –order is ignored.
- This option does not affect the way memory words are split into output files. Think of the files as a set: the set contains a least significant file and a most significant file, but there is no ordering over the set. When you list filenames for a set of files, you *always* list the least significant first, regardless of the –order option.

Figure 10–5 demonstrates how –order affects the conversion process. This figure, and the previous figure, Figure 10–4, explain the condition of the data in the hex conversion utility output files.

Figure 10–5. Varying the Word Order



10.4 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 TMS320C27x linker: both define the memory map of the target address space. Each line entry in the ROMS directive defines a specific address range. The general syntax is:

```
ROMS
{
      romname: [origin=value,] [length=value,] [romwidth=value,]
               [memwidth=value,] [fill=value,]
               [files={filename1, filename2, ...}]

      romname: [origin=value,] [length=value,] [romwidth=value,]
               [memwidth=value,] [fill=value,]
               [files={filename1, filename2, ...}]
   ...
}
```

ROMS begins the directive definition.

romname identifies a memory range. The name of the memory range can be one to eight characters long. 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 10.3.3, *Partitioning Data Into Output Files*, on page 10-9). 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 10.3.2, *Specifying the Memory Width*, on page 10-8). 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 specify the paddr parameter for each section in the SECTIONS directive.* (See section 10.5, *The SECTIONS Directive*, on page 10-20.)

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 10.7.2, *Specifying a Fill Value*, on page 10-25.

files identifies the names of the output files that correspond to this range. List the filenames in order from *least significant* to *most significant* output file, where the bits of the memory word are numbered from right to left. Enclose the list of names in curly braces.

The number of filenames must equal the number of output files that the range generates. To calculate the number of output files generated, see section 10.3.3, *Partitioning Data Into Output Files*, on page 10-9. 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.

10.4.1 When to Use the ROMS Directive

If you do not use a ROMS directive, the utility defines a single default range that includes the entire address space. This is equivalent to a ROMS directive with a single range without origin or length.

Use the ROMS directive when you want to:

- **Program large amounts of data into fixed-size ROMs**. When you specify memory ranges corresponding to the length of your ROMs, the utility automatically breaks the output into blocks that fit into the ROMs.
- **Restrict output to certain segments**. You can also use the ROMS directive to restrict the conversion to a certain segment or segments of the target address space. The utility does not convert the data that falls outside of the ranges defined by the ROMS directive. Sections can span range boundaries; the utility splits them at the boundary into multiple ranges. If a section falls completely outside any of the ranges you define, the utility does not convert that section and issues no messages or warnings. In this way, you can exclude sections without listing them by name with the SECTIONS directive. However, if a section falls partially in a range and partially in unconfigured memory, the utility issues a warning and converts only the part within the range.
- **Use image mode.** When you use the –image option, you must use a ROMS directive. Each range is filled completely so that each output file in a range contains data for the whole range. Holes before, between, or after sections are filled with the fill value from the ROMS directive, with the value specified with the –fill option, or with the default value of 0.

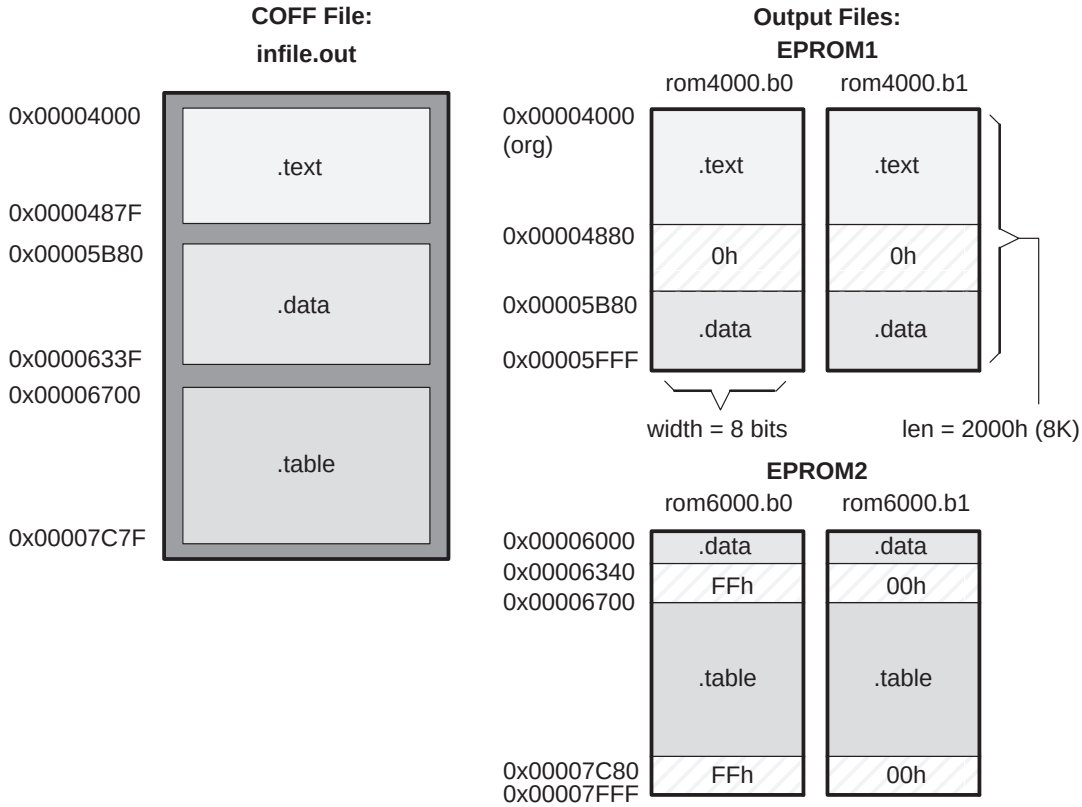
10.4.2 An Example of the ROMS Directive

The ROMS directive in Example 10–1 shows how 16K bytes of 16-bit memory could be partitioned for two 8K × 8-bit EPROMs. Figure 10–6 illustrates the input and output files.

Example 10–1. 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 = 0xFF00,
           files = { rom6000.b0, rom6000.b1}
}
```

Figure 10–6. 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 10–2 is a segment of the map file resulting from Example 10–1.

Example 10–2. Map File Output From Example 10–1 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 = 0000ff00
             00006700..00007c7f .table
             00007c80..00007fff FILL = 0000ff00
```

EPROM1 defines the address range from 0x00004000 through 0x00005FFF. The range contains the following sections:

This section...	Has this range...
.text	0x00004000 through 0x0000487F
.data	0x00005B80 through 0x00005FFF

The rest of the range is filled with 0h (the default fill value). The data from this range is converted into two output files:

- File rom4000.b0 contains bits 0 through 7.
- File rom4000.b1 contains bits 8 through 15.

EPROM2 defines the address range from 0x00006000 through 0x00007FFF. The range contains the following sections:

This section...	Has this range...
.data	0x00006000 through 0x0000633F
.table	0x00006700 through 0x00007C7F

The rest of the range is filled with 0xFF00 (from the specified fill value). The data from this range is converted into two output files:

- File rom6000.b0 contains bits 0 through 7.
- File rom6000.b1 contains bits 8 through 15.

10.5 The SECTIONS Directive

You can convert specific sections of the COFF file by name with the SECTIONS directive. You can also specify those sections that you want to locate in ROM at a different address than the *load* address specified in the linker command file.

- ❏ If you use a SECTIONS directive, the utility converts only the sections that you list in the directive and ignores all other sections in the COFF file.
- ❏ If you do not use a SECTIONS directive, the utility converts all initialized sections that fall within the configured memory. The TMS320C27x C/C++compiler-generated initialized sections include: .text, .const, and .cinit.

Uninitialized sections are *never* converted, whether or not you specify them in a SECTIONS directive.

Note: Sections Generated by the C/C++ Compiler

The TMS320C27x C/C++ compiler automatically generates the following sections.

- ❏ **Initialized sections:** .text, .const, and .cinit.
- ❏ **Uninitialized sections:** .bss, .stack, and .sysmem.

Use the SECTIONS directive in a command file. (For more information, see section 10.2.2, *Invoking the Hex-Conversion Utility With a Command File*, on page 10-5.) The general syntax for the SECTIONS directive is:

```
SECTIONS
{
   sname: [paddr=value],
   sname: [paddr=value],
   ...
}
```

SECTIONS begins the directive definition.

sname identifies a section in the COFF input file. If you specify a section that does not exist, the utility issues a warning and ignores the name.

paddr=*value* specifies the physical ROM address at which this section is to be located. This value overrides the section load address given by the linker. The value must be a decimal, octal, or hexadecimal constant. If one section uses this option, then all sections must use it.

The commas separating section names are optional. For more similarity with the linker's SECTIONS directive, you can use colons after the section names.

Suppose a COFF file contains six initialized sections: .text, .data, .const, .vectors, .coeff, and .tables. If you want only .text and .data to be converted, use a SECTIONS directive as follows:

```
SECTIONS { .text, .data }
```

10.6 Assigning Output Filenames

When the hex-conversion utility translates your COFF object file into a data format, it partitions the data into one or more output files. When multiple files are formed by splitting memory words into ROM words, *filenames are always assigned in order from least to most significant*, where bits in the memory words are numbered from right to left. This is true regardless of target or COFF endian ordering.

The hex-conversion utility follows this sequence when assigning output filenames:

1) **It looks for the ROMS directive.** If a file is associated with a range in the ROMS directive and you have included a list of files (files = {. . .}) in 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 the ROMS directive and –o options are used together, the ROMS directive overrides the –o options.

3) **It assigns a default filename.** If you specify no filenames or fewer names than output files, the utility assigns a default filename. A default filename consists of the base name from the COFF input file plus a 2- to 3-character extension. The extension has three parts:

   a) A format character, based on the output format:

      a for ASCII-Hex
      i for Intel
      t for TI-Tagged
      m for Motorola-S
      x for Tektronix

b) The range number in the ROMS directive. Ranges are numbered starting with 0. If there is no ROMS directive, or only one range, the utility omits this character.

c) The file number in the set of files for the range, starting with 0 for the least significant file.

For example, assume coff.out is a COFF input file for a 16-bit target processor and you are creating Intel format output. With no output filenames specified, the utility produces two output files named coff.i0 and coff.i1.

If you include the following ROMS directive when you invoke the hex-conversion utility, you would have four output files:

```
ROMS
{
   range1: o = 0x00001000 l = 0x1000
   range2: o = 0x00002000 l = 0x1000
}
```

These output files...	Contain data in this location...
coff.i00 and coff.i01	0x00001000 through 0x00001FFF
coff.i10 and coff.i11	0x00002000 through 0x00002FFF

10.7 Image Mode and the –fill Option

This section describes the advantages of operating in image mode and describes how to produce output files with a precise, continuous image of a target memory range.

10.7.1 Generating a Memory Image

With the –image option, the utility generates a memory image by completely filling all of the mapped ranges specified in the ROMS directive.

A COFF file consists of blocks of memory (sections) with assigned memory locations. Typically, all sections are not adjacent: there are holes between sections in the address space for which there is no data. When such a file is converted *without* the use of image mode, the hex-conversion utility bridges these holes by using the address records in the output file to skip ahead to the start of the next section. 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.

Note: Defining the Ranges of Target Memory

If you use image mode, you must also use a ROMS directive. In image mode, each output file corresponds directly to a range of target memory. You must define the ranges. If you do not supply the ranges of target memory, the utility tries to build a memory image of the entire target processor address space, which could create a huge amount of output data. To prevent this situation, the utility requires you to explicitly restrict the address space with the ROMS directive.

10.7.2 Specifying a Fill Value

The –fill option specifies a value for filling the holes between sections. The fill value must be specified as an integer constant following the –fill option. The width of the constant is assumed to be that of a word on the target processor. For example, specifying –fill 0FFh results in a fill pattern of 00FFh. The constant value is not sign extended.

The hex-conversion utility uses a default fill value of 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.

10.7.3 Steps to Follow in Image Mode

Step 1: Define the ranges of target memory with a ROMS directive. See section 10.4, *The ROMS Directive*, on page 10-14 for details.

Step 2: Invoke the hex-conversion utility with the –image option. You can optionally use the –zero option to reset the address origin to 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 value of 0, use the –fill option.

10.8 Controlling the ROM Device Address

The hex-conversion utility output address corresponds to the ROM device address. The EPROM programmer burns the data in the location specified by the address field in the hex-conversion utility output file. The hex-conversion utility offers some mechanisms to control the starting address in ROM of each section. However, many EPROM programmers offer direct control of where the data 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 a hex-conversion utility output file is the load address given in the linker command file.

2) **The paddr option inside the SECTIONS directive.** When the paddr option (described on page 10-20) 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 increments upward, any address record represents offsets from the beginning of the file (the address within ROM) rather than actual target addresses of the data.

   You must use the –zero option in conjunction with the –image option to force the starting address in each output file to be 0. If you specify the –zero option without the –image option, the utility issues a warning and ignores the 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.

10.9 Object Formats

The hex-conversion utility converts a COFF object file into one of five object formats that most EPROM programmers accept as input: ASCII-Hex, Intel MCS-86, Motorola-S, Extended Tektronix, and TI-Tagged.

Table 10–2 specifies the format options.

- 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 10–2. Options for Specifying Hex-Conversion Formats

Option	Format	Address Bits	Default Width
–a	ASCII-Hex	16	8
–i	Intel	32	8
–m	Motorola-S	24	8
–t	TI-Tagged	16	16
–x	Tektronix	32	8

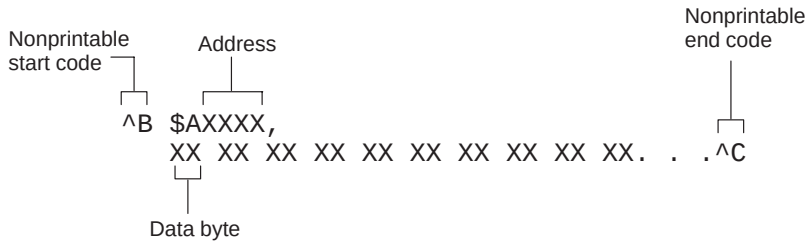
Address bits determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

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.

10.9.1 ASCII-Hex Object Format (–a Option)

The ASCII-Hex object format supports 16-bit addresses. The format consists of a byte stream with bytes separated by spaces. Figure 10–7 illustrates the ASCII-Hex format.

Figure 10–7. ASCII-Hex Object Format



The file begins with an ASCII STX character (ctrl-B, 02h) and ends with an ASCII ETX character (ctrl-C, 03h). Address records are indicated by $AXXXX, in which XXXX is a 4-digit (16-bit) hexadecimal address. The address records are present only in the following situations:

- When discontinuities occur
- When the byte stream does not begin at address 0

You can avoid all discontinuities and any address records by using the –image and –zero options. This creates output that is simply a list of byte values.

10.9.2 Intel MCS-86 Object Format (–i Option)

The Intel object format supports 16-bit addresses and 32-bit extended addresses. It consists of a 9-character (4-field) prefix, the data, and a 2-character checksum suffix. The 9-character prefix defines the start of record, byte count, load address, and record type. These are the record types:

Record Type	Description
00	Data record
01	End-of-file record
04	Extended linear address record

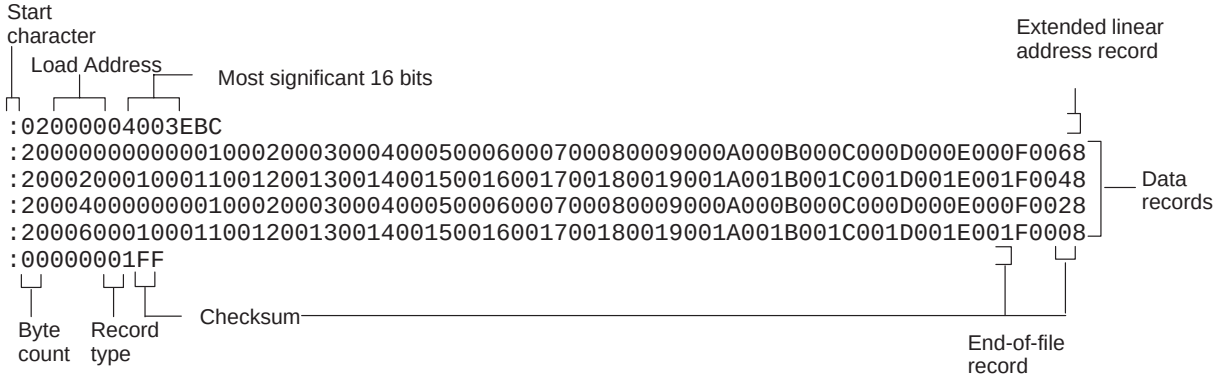
Record type *00*, the data record, begins with a colon (:), which is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 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 (:), which is 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 (:), which is followed by the byte count, a dummy address 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 LSBs of the address.

Figure 10–8 illustrates the Intel hexadecimal object format.

Figure 10–8. Intel MCS86 Hexadecimal Object Format



10.9.3 Motorola-S Object Format (–m Option)

The Motorola-S format supports 24-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
S2	Code/data record
S8	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 10–9 illustrates the Motorola-S object format.

Figure 10–9. Motorola-S Object Format

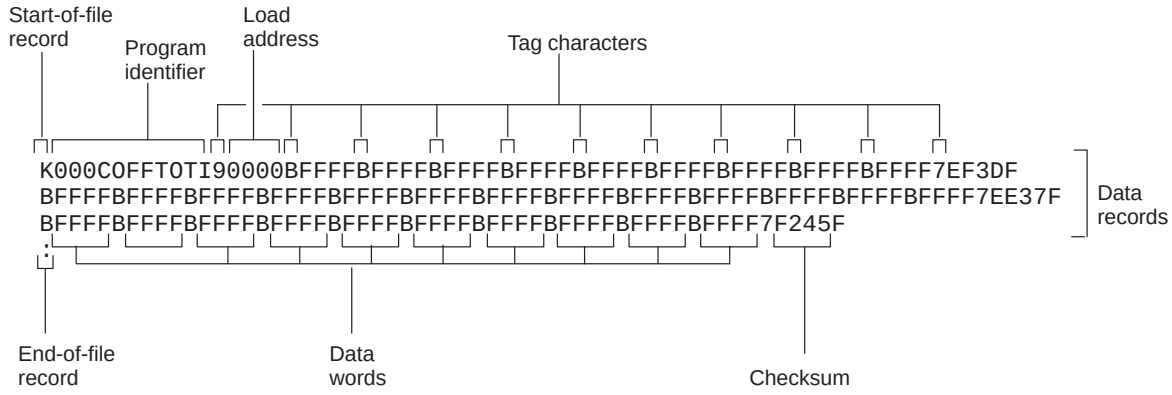
10.9.4 TI-Tagged SDSMAC Object Format (–t Option)

The Texas Instruments SDSMAC (TI-Tagged) object format supports 16-bit addresses. It consists of a start-of-file record, data records, and end-of-file record. Each data record consists of a series of small fields and is signified by a tag character. The significant tag characters are:

Tag Character	Description
K	Followed by the program identifier
7	Followed by a checksum
8	Followed by a dummy checksum (ignored)
9	Followed by a 16-bit load address
B	Followed by a data word (four characters)
F	Identifies the end of a data record
*	Followed by a data byte (two characters)

Figure 10–10 illustrates the tag characters and fields in TI-Tagged object format.

Figure 10–10. TI-Tagged Object Format



If any data fields appear before the first address, the first field is assigned address 0000h. Address fields may be expressed for any data byte, but none is required. The checksum field, which is preceded by the tag character 7, is a 2s complement of the sum of the 8-bit ASCII values of characters, beginning with the first tag character and ending with the checksum tag character (7 or 8). The end-of-file record is a colon (:).

10.9.5 Extended Tektronix Object Format (–x Option)

The Tektronix object format supports 32-bit addresses and has two types of records:

Data records contain the header field, the load address, and the object code.

Termination records signify the end of a module.

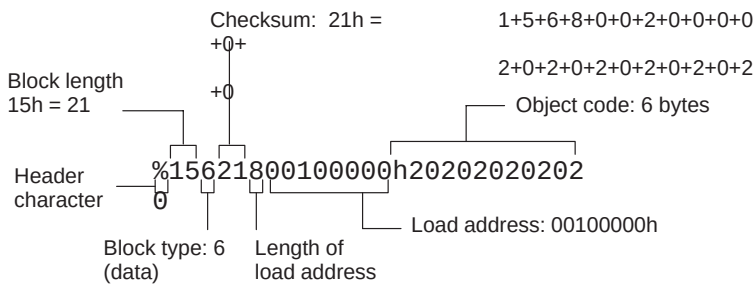
The header field in the data record contains the following information:

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 is to be located. The first digit specifies the address length. The remaining characters of the data record contain the object code in the form of two characters per byte.

Figure 10–11 illustrates the Tektronix object format.

Figure 10–11. Extended Tektronix Object Format



10.10 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 *TMS320C27x 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 10.3, *Understanding Memory Widths*, on page 10-7 and section 10.8, *Controlling the ROM Device Address*, on page 10-26.

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.

Appendix A

Common Object File Format

The TMS320C27x™ assembler and linker create object files in common object file format (COFF). COFF is an implementation of an object file format of the same name that was developed by AT&T for use on UNIX based systems. This format is used because it encourages modular programming and provides powerful and flexible methods for managing code segments and target system memory.

Sections are a basic COFF concept. Chapter 2, *Introduction to Common Object File Format*, discusses COFF sections in detail. If you understand section operation, you can use the assembly language tools more efficiently.

This appendix contains technical details about COFF object file structure. Much of this information pertains to the symbolic debugging information that is produced by the C compiler. The purpose of this appendix is to provide supplementary information on the internal format of COFF object files.

Topic **Page**

A.1 COFF File Structure

The elements of a COFF object file describe the file's sections and symbolic debugging information. These elements include:

- A file header
- Optional header information
- A table of section headers
- Raw data for each initialized section
- Relocation information for each initialized section
- Line-number entries for each initialized section
- A symbol table
- A string table

The assembler and linker produce object files with the same COFF structure; however, a program that is linked for the final time does not usually contain relocation entries. Figure A–1 illustrates the object file structure.

Figure A–1. COFF File Structure

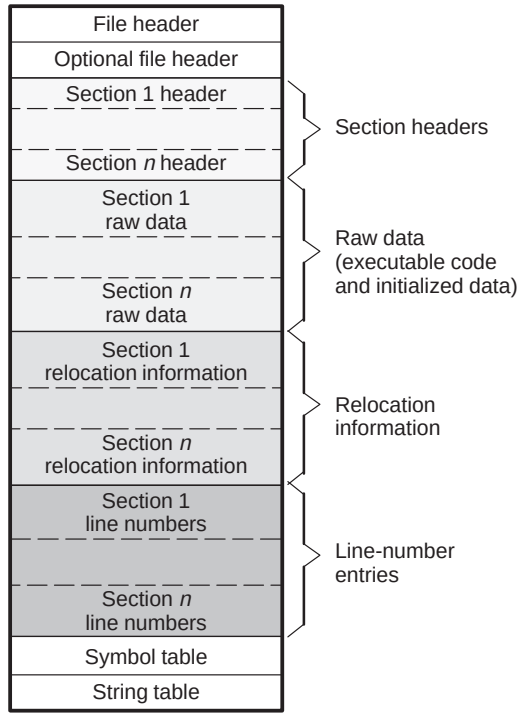
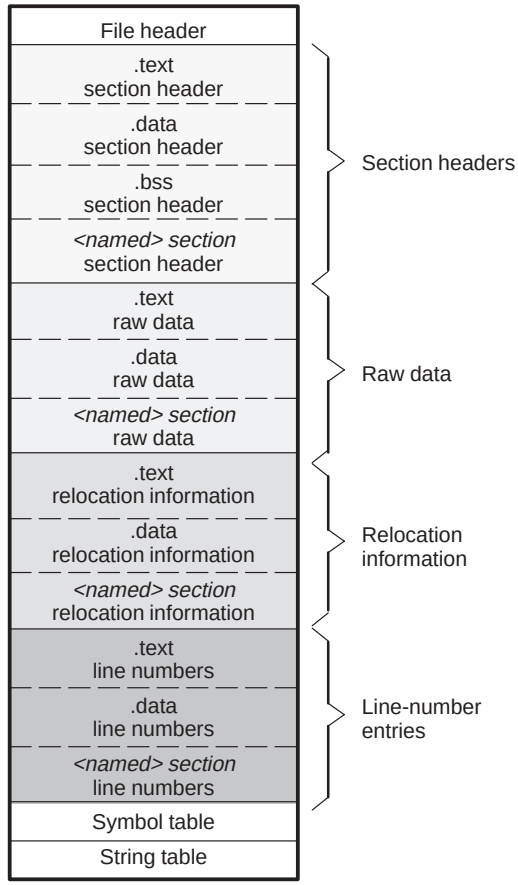


Figure A–2 shows a typical example of a COFF object file that contains the three default sections, .text, .data, and .bss, and a named section (referred to as <named>). By default, the tools place sections into the object file in the following order: .text, .data, initialized named sections, .bss, and uninitialized named sections. Although uninitialized sections have section headers, they have no raw data, relocation information, or line number entries. This is because the .bss and .usect directives simply reserve space for uninitialized data; uninitialized sections contain no actual code.

Figure A–2. Sample COFF Object File



A.2 File Header Structure

The file header contains 22 bytes of information that describe the general format of an object file. Table A–1 shows the structure of the TMS320C27x COFF file header.

Table A–1. File Header Contents

Byte Number	Type	Description
0–1	Unsigned short	Version ID; indicates the version of the COFF file structure
2–3	Unsigned short	Number of section headers
4–7	Integer	Time and date stamp; indicates when the file was created
8–11	Integer	File pointer; contains the symbol table's starting address
12–15	Integer	Number of entries in the symbol table
16–17	Unsigned short	Number of bytes in the optional header. This field is either 0 or 28; if it is 0, there is no optional file header.
18–19	Unsigned short	Flags (see Table A–2)
20–21	Unsigned short	Target ID; magic number (00ebh) indicates the file can be executed in a TMS320C27x system.

Table A–2 lists the flags that can appear in bytes 18 and 19 of the file header. Any number and combination of these flags can be set at the same time (for example, if bytes 18 and 19 are set to 0003h, both F_RELFLG and F_EXEC are set.

Table A–2. File Header Flags (Bytes 18 and 19)

Mnemonic	Flag	Description
F_RELFLG	0001h	Relocation information was stripped from the file.
F_EXEC	0002h	The file is executable (it contains no unresolved external references).
F_LNNO	0004h	Line numbers were stripped from the file.
F_LSYMS	0008h	Local symbols were stripped from the file.
F_LITTLE	0100h	The target is a little-endian device.
F_BIG	0200h	The target is a big-endian device.

A.3 Optional File Header Format

The linker creates the optional file header and uses it to perform relocation at load time. Partially linked files do not contain optional file headers. Table A–3 illustrates the optional file header format.

Table A–3. Optional File Header Contents

Byte Number	Type	Description
0–1	Short	Optional file header magic number (0108h)
2–3	Short	Version stamp
4–7	Integer	Size (in bytes) of executable code
8–11	Integer	Size (in bytes) of initialized data
12–15	Integer	Size (in bytes) of uninitialized data
16–19	Integer	Entry point
20–23	Integer	Beginning address of executable code
24–27	Integer	Beginning address of initialized data

A.4 Section Header Structure

COFF object files contain a table of section headers that define where each section begins in the object file. Each section has its own section header. Table A–4 shows the structure of each section header.

Table A–4. Section Header Contents

Byte Number	Type	Description
0–7	Character	This field contains one of the following: ❑ An 8-character section name, padded with nulls ❑ A pointer into the string table if the symbol name is longer than eight characters
8–11	Integer	Section's physical address
12–15	Integer	Section's virtual address
16–19	Integer	Section's size in words
20–23	Integer	File pointer to raw data
24–27	Integer	File pointer to relocation entries
28–31	Integer	File pointer to line-number entries
32–35	Unsigned integer	Number of relocation entries
36–39	Unsigned integer	Number of line-number entries
40–43	Unsigned integer	Flags (see Table A–5)
44–45	Unsigned short	Reserved
46–47	Unsigned short	Memory page number

Table A–5 lists the flags that can appear in bytes 36 through 39 of the section header.

Table A–5. Section Header Flags (Bytes 40 Through 43)

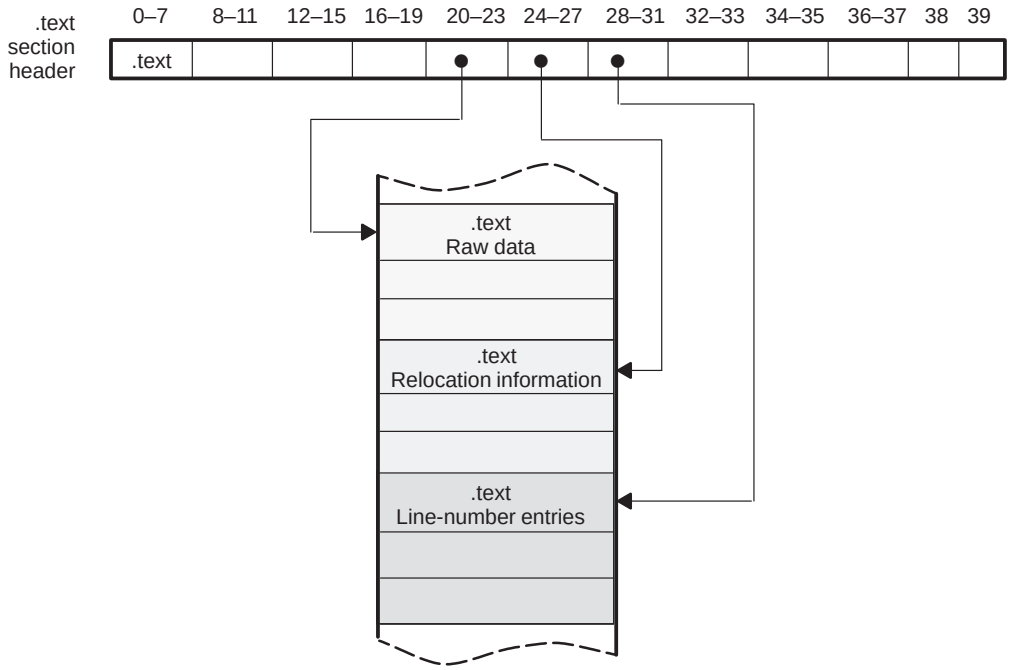
Mnemonic	Flag	Description
STYP_REG	00000000h	Regular section (allocated, relocated, loaded)
STYP_DSECT	00000001h	Dummy section (relocated, not allocated, not loaded)
STYP_NOLOAD	00000002h	Noload section (allocated, relocated, not loaded)
STYP_GROUP	00000004h	Grouped section (formed from several input sections)
STYP_PAD	00000008h	Padding section (loaded, not allocated, not relocated)
STYP_COPY	00000010h	Copy section (relocated, loaded, but not allocated; relocation and line-number entries are processed normally)
STYP_TEXT	00000020h	Section contains executable code
STYP_DATA	00000040h	Section contains initialized data
STYP_BSS	00000080h	Section contains uninitialized data
STYP_CLINK	00004000h	Section requires conditional linking

Note: The term *loaded* means that the raw data for this section appears in the object file.

The flags listed in Table A–5 can be combined; for example, if the flag's word is set to 024h, both STYP_GROUP and STYP_TEXT are set.

Figure A–3 illustrates how the pointers in a section header points to the elements in an object file that are associated with the .text section.

Figure A–3. Section Header Pointers for the .text Section



As Figure A–2 on page A-3 shows, uninitialized sections (created with the .bss and .usect directives) vary from this format. Although uninitialized sections have section headers, they have no raw data, relocation information, or line-number information. They occupy no actual space in the object file. Therefore, the number of relocation entries, the number of line-number entries, and the file pointers are 0 for an uninitialized section. The header of an uninitialized section simply tells the linker how much space for variables it should reserve in the memory map.

A.5 Structuring Relocation Information

A COFF object file has one relocation entry for each relocatable reference. The assembler automatically generates relocation entries. The linker reads the relocation entries as it reads each input section and performs relocation. The relocation entries determine how references within each input section are treated.

COFF file relocation information entries use the 10-byte format summarized in Table A–6. the fields in each entry are described following the table.

Table A–6. Relocation Entry Contents

Byte Number	Type	Description
0–3	Integer	Virtual address of the reference
4–5	Unsigned short	Symbol-table index (0–65535)
6–7	Unsigned short	Reserved
8–9	Unsigned short	Relocation type (see Table A–7)

- The **virtual address** is the symbol's address in the current section *before* relocation; it specifies *where* a relocation must occur. (This is the address of the field in the object code that must be patched.)

Following is an example of code that generates a relocation entry:

```
2                       .global   X
3   000000 ffff!        b         X
    000001 0000
```

In this example, the virtual address of the relocatable field is 0001.

- The **symbol-table index** is the index of the referenced symbol. In the preceding example, this field would contain the index of X in the symbol table. The amount of the relocation is the difference between the symbol's current address in the section and its assembly-time address. The relocatable field must be relocated by the same amount as the referenced symbol. In the example, X has a value of 0h before relocation. Suppose X is relocated to address 2000h. This is the relocation amount (2000h – 0h = 2000h), so the relocation field at address 1 is patched by adding 2000h to it.

You can determine a symbol's relocated address if you know the section in which it is defined. For example, if X is defined in .data and .data is relocated by 2000h, X is relocated by 2000h.

If the symbol table index in a relocation entry is –1 (0FFFFh), the relocation is called an *internal relocation.* In this case, the relocation amount is simply the amount by which the current section is being relocated.

- The **relocation type** specifies the size of the field to be patched and describes how the patched value is calculated. The type field depends on the addressing mode that was used to generate the relocatable reference. In the preceding example, the actual address of the referenced symbol X is a 22-bit address, but a 16-bit relative address is coded in to the object code. The 22-bit address is used to find the offset of the jump from this location. Thus, the relocation type is R_C27PCR16. Table A–7 lists the relocation types.

Table A–7. Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_RELBYTE	000Fh	8-bit direct reference to symbol's address
R_RELWORD	0010h	16-bit direct reference to symbol's address
R_RELLONG	0011h	32-bit direct reference to symbol's address
R_PARTLS6	05Dh	6-bit offset of a 22-bit address
R_PARTLS7	28h	7-bit offset of a 16–bit address
R_PARTMID10	05Eh	Middle 10 bits of a 22-bit address
R_REL22	05Fh	22-bit direct reference to symbol's address
R_PARTMS6	060h	Upper 6 bits of a 22-bit address
R_PARTS16	061h	Upper 16 bits of a 22-bit address
R_C27PCR16	062h	PC relative 16-bit
R_C27PCR8	063h	PC relative 8-bit
R_C27PTR	064h	22-bit pointer
R_C27HI16	065h	High 16 bits of address data
R_C27LOPTR	066h	Pointer to low 64K
R_C27NWORD	067h	16-bit negated relocation
R_C27NBYTE	068h	8-bit negated relocation

A.6 Line-Number Table Structure

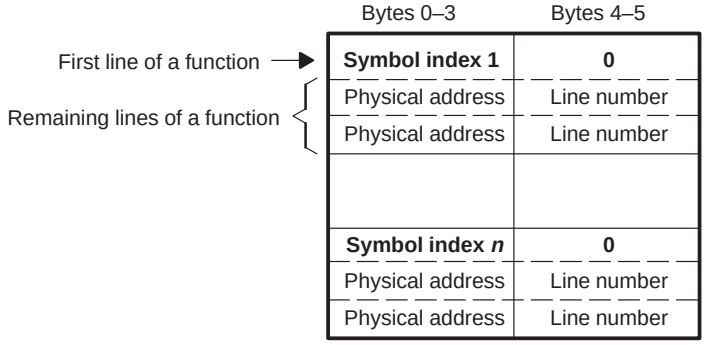
The object file contains a table of line number entries that are useful for symbolic debugging. When the C compiler produces several lines of assembly language code, it creates a line-number entry that maps these lines back to the original line of C source code that generated them. Each single line-number entry contains six bytes of information. Table A–8 shows the format of a line-number entry.

Table A–8. Line-Number Entry Format

Byte Number	Type	Description
0–3	Integer	This entry can have one of two values: ❑ If it is the first entry in a block of line-number entries, the value is an index that points to a symbol entry in the symbol table. ❑ If it is not the first entry in a block, it is the physical address of the line indicated by bytes 4–5.
4–5	Unsigned short	This entry may have one of two values: ❑ If the value of this field is 0, this is the first line of a function entry. ❑ If the value of this field is *not* 0, this is the line number of a line of C source code.

Figure A–4 shows how line-number entries are grouped into blocks.

Figure A–4. Line-Number Blocks



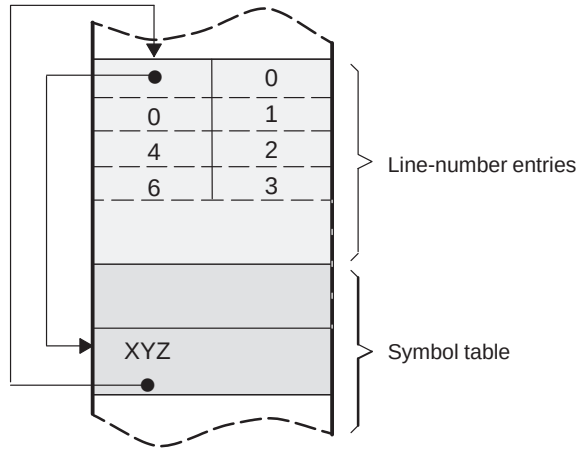
As Figure A–4 shows, each entry is divided into halves:

- For the *first line* of a function, bytes 0–3 point to the name of a symbol or a function in the symbol table, and bytes 4–5 contain a 0, which indicates the beginning of a block.
- For the *remaining lines* in a function, bytes 0–3 show the physical address (the number of bytes created by a line of C source), and bytes 4–5 show the address of the original C source, relative to its appearance in the C source program.

The line-number entry table can contain many of these blocks.

Figure A–5 illustrates line-number entries for a function named XYZ. As shown, the function name is entered as a symbol in the symbol table. The first portion on XYZ's block of line-number entries points to the function name in the symbol table. Assume that the original function in the C source contained three lines of code. The code associated with the first line is located at byte offset 0 from the beginning of the function. The code for line 2 begins at offset 4, and the code associated with line 3 is 6 bytes from the beginning of the function.

Figure A–5. Line Number Entries



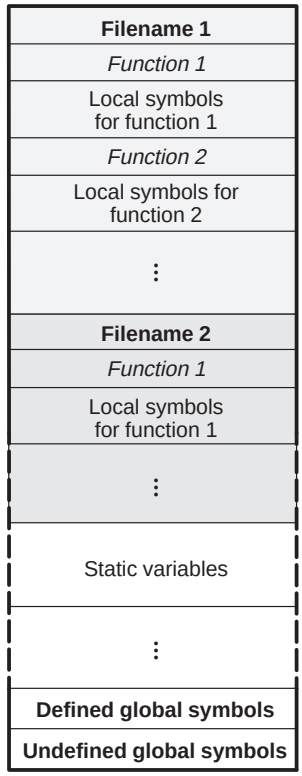
(The symbol-table entry for XYZ has a field that points back to the beginning of the line-number block.)

Because line numbers are not often needed, the linker provides an option (–s) that strips line-number information from the object file; this provides a more compact object module. (For more information on the –s option, see section 7.4.15, *Strip Symbolic Information (–s Option)*, page 7-18.)

A.7 Symbol Table Structure and Content

The order of symbols in the symbol table is very important; they appear in the sequence shown in Figure A–6.

Figure A–6. Symbol-Table Contents



Static variables refer to symbols defined in C that have storage-class static outside any function. If you have several modules that use symbols with the same name, making them static confines the scope of each symbol to the module that defines it (this eliminates multiple-definition conflicts).

The entry for each symbol in the symbol table contains the symbol's:

- ❏ Name (or an offset into the string table)
- ❏ Type
- ❏ Value
- ❏ Section it was defined in
- ❏ Storage class
- ❏ Basic type (integer, character, etc.)
- ❏ Derived type (array, structure, etc.)
- ❏ Dimensions
- ❏ Line number of the source code that defined the symbol

Section names are also defined in the symbol table.

All symbol entries, regardless of class and type, have the same format in the symbol table. Each symbol table entry contains the 18 bytes of information listed in Table A–9. Each symbol may also have an 18-byte auxiliary entry; the special symbols listed in Table A–10, page A-16, always have an auxiliary entry. Some symbols may not have all the characteristics specified in the preceding list; if a particular field is not set, it is set to null.

Table A–9. Symbol-Table Entry Contents

Byte Number	Type	Description
0–7	Char	This field contains one of the following: ❏ An 8-character symbol name, padded with nulls ❏ A pointer into the string table if the symbol name is longer than eight characters
8–11	Integer	Symbol value; storage class dependent
12–13	Short	Section number of the symbol
14–15	Unsigned short	Basic and derived type specification
16	Char	Storage class of the symbol
17	Char	Number of auxiliary entries (always 0 or 1)

A.7.1 Special Symbols

The symbol_table contains some special symbols that are generated by the compiler, assembler, and linker. Each special symbol contains ordinary symbol table information as well as an auxiliary entry. Table A–10 lists these symbols.

Several of these symbols appear in pairs:

- The .bb/.eb symbols indicate the beginning and end of a block.
- The .bf/.ef symbols indicate the beginning and end of a function.
- The *n*fake/.eos symbols name and define the limits of structures, unions, and enumerations that were not named. The .eos symbol is also paired with named structures, unions, and enumerations.

Table A–10. Special Symbols in the Symbol Table

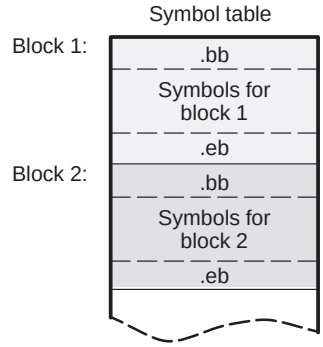
Symbol	Description
.text	Address of the .text section
.data	Address of the .data section
.bss	Address of the .bss section
.bb	Address of the beginning of a block
.eb	Address of the end of a block
.bf	Address of the beginning of a function
.ef	Address of the end of a function
.target	Pointer to a structure or union that is returned by a function
.*n*fake†	Dummy tag name for a structure, union, or enumeration
.eos	End of a structure, union, or enumeration
etext	Next available address after the end of the .text output section
edata	Next available address after the end of the .data output section
end	Next available address after the end of the .bss output section

† When a structure, union, or enumeration has no tag name, the compiler assigns it a name so that it can be entered into the symbol table. These names are of the form *n*fake, where *n* is an integer. The compiler begins numbering these symbol names at 0.

A.7.1.1 Symbols and Blocks

In C, a block is a compound statement that begins and ends with braces. A block always contains symbols. The symbol definitions for any particular block are grouped together in the symbol table and are delineated by the .bb/.eb special symbols. Blocks can be nested in C, and their symbol table entries can be nested correspondingly. Figure A–7 shows how block symbols are grouped in the symbol table.

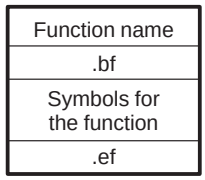
Figure A–7. Symbols for Blocks



A.7.1.2 Symbols and Functions

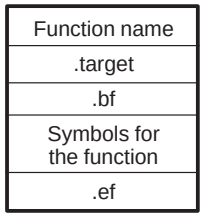
The symbol definitions for a function appear in the symbol table as a group, delineated by .bf/.ef special symbols. The symbol table entry for the function name precedes the .bf special symbol. Figure A–8 shows the format of symbol table entries for a function.

Figure A–8. Symbols for Functions



If a function returns a structure or union, a symbol-table entry for the special symbol .target appears between the entries for the function name and the .bf special symbol, as shown in Figure A–9.

Figure A–9. Symbols for Functions That Return a Structure or Union



A.7.2 Symbol Name Format

The first eight bytes of a symbol-table entry (bytes 0–7) indicate a symbol's name:

- If the symbol name is eight characters or less, this field has type *character.* The name is padded with nulls (if necessary) and stored in bytes 0–7.
- If the symbol name is greater than eight characters, this field is treated as two integers. The entire symbol name is stored in the string table. Bytes 0–3 contain 0, and bytes 4–7 are an offset into the string table.

A.7.3 String Table Structure

Symbol names that are longer than eight characters are stored in the string table. The field in the symbol-table entry that would normally contain the symbol's name contains, instead, a pointer to the symbol's name in the string table. Names are stored contiguously in the string table, delimited by a null byte. The first four bytes of the string table contain the size of the string table in bytes; thus, offsets into the string table are greater than or equal to 4.

Figure A–10 is a string table that contains two symbol names, *Adaptive-Filter* and *Fourier-Transform*. The index in the string table is 4 for Adaptive-Filter and 20 for Fourier-Transform.

Figure A–10. String-Table Entries for Sample Symbol Names.

38 bytes

4 bytes			
'A'	'd'	'a'	'p'
't'	'i'	'v'	'e'
'-'	'F'	'i'	'l'
't'	'e'	'r'	'\0'
'F'	'o'	'u'	'r'
'i'	'e'	'r'	'-'
'T'	'r'	'a'	'n'
's'	'f'	'o'	'r'
'm'	'\0'		

A.7.4 Storage Classes

Byte 16 of the symbol-table entry indicates the storage class of the symbol. Storage classes refer to the method in which the C compiler accesses a symbol. Table A–11 lists valid storage classes.

Table A–11. Symbol Storage Classes

Mnemonic	Value	Storage Class	Mnemonic	Value	Storage Class
C_NULL	0	No storage class	C_USTATIC	14	Undefined static
C_AUTO	1	Automatic variable	C_ENTAG	15	Enumeration tag
C_EXT	2	External definition	C_MOE	16	Member of an enumeration
C_STAT	3	Static	C_REGPARM	17	Register parameter
C_REG	4	Register variable	C_FIELD	18	Bit field
C_EXTREF	5	External reference	C_UEXT	19	Tentative external definition
C_LABEL	6	Label	C_STATLAB	20	Static load time label
C_ULABEL	7	Undefined label	C_EXTLAB	21	External load time label
C_MOS	8	Member of a structure	C_BLOCK	100	Beginning or end of a block; used only for the .bb and .eb special symbols
C_ARG	9	Function argument	C_FCN	101	Beginning or end of a function; used only for the .bf and .ef special symbols
C_STRTAG	10	Structure tag	C_EOS	102	End of structure; used only for the .eos special symbol
C_MOU	11	Member of a union	C_FILE	103	Filename; used only for filename symbols
C_UNTAG	12	Union tag	C_LINE	104	Used only by utility programs
C_TPDEF	13	Type definition			

Some special symbols are restricted to certain storage classes. Table A–12 lists these symbols and their storage classes.

Table A–12. Special Symbols and Their Storage Classes

Special Symbol	Restricted to This Storage Class	Special Symbol	Restricted to This Storage Class
.bb	C_BLOCK	.eos	C_EOS
.eb	C_BLOCK	.text	C_STAT
.bf	C_FCN	.data	C_STAT
.ef	C_FCN	.bss	C_STAT

A.7.5 Symbol Values

Bytes 8–11 of a symbol-table entry indicate a symbol's value. A symbol's value depends on the symbol's storage class; Table A–13 summarizes the storage classes and related values.

Table A–13. Symbol Values and Storage Classes

Storage Class	Value Description	Storage Class	Value Description
C_AUTO	Stack offset in bits	C_UNTAG	0
C_EXT	Relocatable address	C_TPDEF	0
C_STAT	Relocatable address	C_ENTAG	0
C_REG	Register number	C_MOE	Enumeration value
C_LABEL	Relocatable address	C_REGPARM	Register number
C_MOS	Offset in bits	C_FIELD	Bit displacement
C_ARG	Stack offset in bits	C_BLOCK	Relocatable address
C_STRTAG	0	C_FCN	Relocatable address
C_MOU	Offset in bits	C_FILE	0

The value of a relocatable symbol is its virtual address. When the linker relocates a section, the value of a relocatable symbol changes accordingly.

A.7.6 Section Number

Bytes 12–13 of a symbol table entry contain a number that indicates the section in which the symbol was defined. Table A–14 lists these numbers and the sections they indicate.

Table A–14. Section Numbers

Mnemonic	Section Number	Description
N_DEBUG	–2	Special symbolic debugging symbol
N_ABS	–1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1	.text section (typical)
N_SCNUM	2	.data section (typical)
N_SCNUM	3	.bss section (typical)
N_SCNUM	4–32 767	Section number of a named section, in the order in which the named sections are encountered

If there were no .text, .data, or .bss sections, the numbering of named sections would begin with 1.

If a symbol has a section number of 0, –1, or –2, it is not defined in a section. A section number of –2 indicates a symbolic debugging symbol, which includes structure, union, and enumeration tag names, type definitions, and the filename. A section number of –1 indicates that the symbol has a value but is not relocatable. A section number of 0 indicates a relocatable external symbol that is not defined in the current file.

A.7.7 Type Entry

Bytes 14–15 of the symbol table entry define the symbol's type. Each symbol has one basic type and one to six derived types.

Following is the format for this 16-bit type entry:

	Derived type 6	Derived type 5	Derived type 4	Derived type 3	Derived type 2	Derived type 1	Basic type
Size (in bits):	2	2	2	2	2	2	4

Bits 0–3 of the type field indicate the basic type. Table A–15 lists valid basic types.

Table A–15. Basic Types

Mnemonic	Value	Type
T_VOID	0	Void type
T_SCHAR1	1	Character (explicitly signed)
T_CHAR	2	Character (implicitly signed)
T_SHORT	3	Short
T_INT	4	Integer
T_LONG	5	Integer
T_FLOAT	6	Floating point
T_DOUBLE	7	Double floating point
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_LDOUBLE	11	Long double floating point
T_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short
T_UINT	14	Unsigned integer
T_ULONG	15	Unsigned integer

Bits 4–15 of the type field are arranged as six 2-bit fields, each of which can indicate a derived type. Table A–16 lists the derived types.

Table A–16. Derived Types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARY	3	Array

An example of a symbol with several derived types is a symbol with a type entry of 0000 0000 1101 0011_2. This entry indicates that the symbol is an array of pointers to shorts.

A.7.8 Auxiliary Entries

Each symbol table entry can have *one* or *no* auxiliary entry. An auxiliary symbol-table entry contains the same number of bytes as a symbol table entry (18), but the format of an auxiliary entry depends on the symbol's type and storage class. Table A–17 summarizes these relationships.

Table A–17. Auxiliary Symbol-Table Entries Format

		Type Entry		
Name	**Storage Class**	**Derived Type 1**	**Basic Type**	**Auxiliary Entry Format**
.text, .data, .bss	C_STAT	DT_NON	T_VOID	Section (see Table A–18)
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_STRUCT T_UNION T_ENUM	Tag name (see Table A–19)
.eos	C_EOS	DT_NON	T_VOID	End of structure (see Table A–20)
fcname	C_EXT C_STAT	DT_FCN	Any	Function (see Table A–21)
arrname	See note 1	DT_ARY	See note 2	Array (see Table A–22)
.bb, .eb	C_BLOCK	DT_NON	T_VOID	Beginning and end of a block (see Table A–23 and Table A–24)
.bf, .ef	C_FCN	DT_NON	T_VOID	Beginning and end of a function (see Table A–23 and Table A–24)
Name related to a structure, union, or enumeration	See note 1	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION T_ENUM	Name related to a structure, union, or enumeration (see Table A–25)

Notes: 1) C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF, C_EXT
2) Any except T_VOID

In Table A–17, *tagname* refers to any symbol name (including the special symbol *.*fake); *fcname* and *arrname* also refer to any symbol name. Typically, tagname refers to a structure, fcname refers to a function, and arrname refers to an array.

A symbol that satisfies more than one condition in Table A–17 must have a union format in its auxiliary entry. A symbol that satisfies none of these conditions cannot have an auxiliary entry.

A.7.8.1 Sections

Table A–18 illustrates the format of auxiliary table entries.

Table A–18. Section Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Integer	Section length
4–5	Unsigned short	Number of relocation entries
6–7	Unsigned short	Number of line number entries
8–17	—	Not used (zero filled)

A.7.8.2 Tag Names

Table A–19 illustrates the format of auxiliary table entries for tag names.

Table A–19. Tag Name Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	—	Unused (zero filled)
4–7	Integer	Size of structure, union, or enumeration
8–11	—	Unused (zero filled)
12–15	Integer	Index of next entry beyond this function
16–17	—	Unused (zero filled)

A.7.8.3 End of Structure

Table A–20 illustrates the format of auxiliary table entries for ends of structures.

Table A–20. End-of-Structure Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Integer	Tag index
4–7	Integer	Size of structure, union, or enumeration
8–17	—	Unused (zero filled)

A.7.8.4 Functions

Table A–21 illustrates the format of auxiliary table entries for functions.

Table A–21. Function Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Integer	Tag index
4–7	Integer	Size of function (in bits)
8–11	Integer	File pointer to line number
12–15	Integer	Index of next entry beyond this function
16–17	—	Unused (zero filled)

A.7.8.5 Arrays

Table A–22 illustrates the format of auxiliary table entries for arrays.

Table A–22. Array Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Integer	Tag index
4–7	Integer	Size of array
8–9	Unsigned short	First dimension
10–11	Unsigned short	Second dimension
12–13	Unsigned short	Third dimension
14–15	Unsigned short	Fourth dimension
16–17	—	Unused (zero filled)

A.7.8.6 End of Blocks and Functions

Table A–23 illustrates the format of auxiliary table entries for the ends of blocks and functions.

Table A–23. End-of-Blocks/Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	—	Unused (zero filled)
4–5	Unsigned short	C source line number
6–17	—	Unused (zero filled)

A.7.8.7 Beginning of Blocks and Functions

Table A–24 illustrates the format of auxiliary table entries for the beginnings of blocks and functions.

Table A–24. Beginning-of-Blocks/Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Integer	Register save mask
4–5	Unsigned short	C source line number of block begin
6–7	Unsigned short	Number of line entries for function
8–11	Integer	Size of local frame for function
12–15	Integer	Index of next entry past this block
16–17	—	Unused (zero filled)

A.7.8.8 Names Related to Structures, Unions, and Enumerations

Table A–25 illustrates the format of auxiliary table entries for the names of structures, unions, and enumerations.

Table A–25. Structure, Union, and Enumeration Names Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Integer	Tag index
4–7	Integer	Size of the structure, union, or enumeration
8–17	—	Unused (zero filled)

Appendix B

Symbolic Debugging Directives

The assembler supports several directives that the TMS320C27x C/C++ compiler uses for symbolic debugging:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate 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 function.
- The **.block** and **.endblock** directives specify the bounds of C blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source filename.
- The **.line** directive identifies the line number of a C source statement.

These symbolic debugging directives are not usually listed in the assembly language file that the compiler creates. If you want them to be listed and you want to retain the assembly language file, invoke the compiler shell with the –g and –k options, as shown below:

```
cl2000 –gk input file
```

This appendix contains an alphabetical directory of the symbolic debugging directives. With the exception of the .file directive description, each directive contains an example of C source and the resulting assembly language code.

For information on the C compiler, refer to the *TMS320C27x Optimizing C/C++ Compiler User's Guide*.

Syntax

.block [*beginning line number*]
.endblock [*ending line number*]

Description

The **.block** and **.endblock** directives specify the beginning and end of a C block. The *line numbers* are optional; they specify the location in the source file where the block is defined.

Block definitions can be nested. The assembler detects improper block nesting.

Example

Following is an example of C source that defines a block and the resulting assembly language code.

C source:

```
main()
{
   int i = 10;
   {
      int y = i + 3;
      foo(y);
   }
}
```

Resulting assembly language code:

```
_main:
     .sym    _i,-1,4,1,16
     .sym    _y,-2,4,1,16
     ADDB    SP,#2
     .line   3
     MOVB        AL,#10        ;|5|
     MOV         *-SP[1],AL    ;|5|
     .block  7
     .line   5
     ADDB        AL,#3         ;|7|
     MOV         *-SP[2],AL    ;|7|
     .line   6
     LC      #_foo             ;|8|
     ; call occurs [#_foo]     :|8|
     .endblock   9
     SUBB        SP,#2
     LRET
     ; return occurs
     .endfunc    11,000000000h,2
```

Syntax

.file "*filename***"**

Description

The **.file** directive allows a debugger to map locations in memory back to lines in a C source file. The *filename* is the name of the file that contains the original C source program. Filenames can be arbitrarily long.

You can also use the .file directive in assembly code to provide a name in the file and improve program readability.

Example

In the following example, the file named text.c contained the C source that produced this directive.

```
.file     "text.c"
```

Syntax

.func [*beginning line number*]
.endfunc [*ending line number*]

Description

The **.func** and **.endfunc** directives specify the beginning and end of a C function. The *line numbers* are optional; they specify the location in the source file where the function is defined. Function definitions cannot be nested.

Example

Following is an example of C source that defines a function and the resulting assembly language code. The assembly language code is long due to optimization, but software pipelining enables the use of parallel instructions that cause the code to execute fast.

C source:

```
power(x, n)    /*  Beginning of a function */
int x,n;
{
   int i, p;
   p = 1;
   for (i =1; i <= n; ++i)
      p = p *x;
   return p;   /*  End of a function       */
}
```

Resulting assembly language code:

```
       .func  3
_power:
;*AL   assigned to _x
       .sym   _x,0,4,17,16
;*AH   assigned to _n
       .sym   _n,1,4,17,16
       .sym   _x,-1,4,1,16
       .sym   _n,-2,4,1,16
       .sym   _i,-3,4,1,16
       .sym   _p,-4,4,1,16
       ADDB       SP,#4
       .line  2
       MOV        *-SP[1],AL    ;|4|
       MOV        *-SP[2],AH    ;|4|
       .line  5
       MOVB       AL,#1         ;|7|
       MOV        *-SP[4],AL    ;|7|
       .line  7
       MOV        *-SP[3],AL    ;|9|
       MOV        AL,*-SP[2]    ;|9|
       CMP        AL,*-SP[3]    ;|9|
       B          L3,LT         ;|9|
       branch occurs            ;|9|

L2:
       .line  8
       MOV        T,*-SP[1]     ;|10|
       MPY        ACC,T,*-SP[4] ;|10|
       MOV        *-SP[4],AL    ;|10|
       .line  7
       INC        *-SP[3]       ;|9|
       MOV        AL,*-SP[2]    ;|9|
       CMP        AL,*-SP[3]    ;|9|
       B          L2,GEQ        ;|9|
       ;branch occurs           ;|9|
L3:
       .line  9
       MOV        AL,*-SP[4]    ;|11|
       .line  10
       SUBB       SP,#4         ;|12|
       LRET
       ;return occurs
       .endfunc     12,000000000h,4
```

Syntax

.line *line number* [**,** *address*]

Description

The **.line** directive creates a line-number entry in the object file. Line-number entries are used in symbolic debugging to associate addresses in the object code with the lines in the source code that generated them.

The .line directive has two operands:

- The *line number* indicates the line of the C source that generated a portion of code. Line numbers are relative to the beginning of the current function. This is a required parameter.
- The *address* is an expression that is the address associated with the line number. This is an optional parameter; if you do not specify an address, the assembler uses the current SPC value.

Example

The .line directive is followed by the assembly language source statements that are generated by the indicated line of C source. For example, assume that the following lines of C are lines 4 through 6 in the original C source; line 5 produces the assembly language source statements that are shown below.

C source:

```
for (i = 0; i <= 5; ++i)
   sum = sum + i;
return sum;
```

Resulting assembly language code:

```
_main:
       .sym   _i,-1,4,1,16
       .sym   _sum,-2,4,1,16
       ADDB       SP,#2
       .line  3
       MOV        *-SP[2],#0       ;|5|
       .line  5
       MOV        *-SP[1],#0       ;|7|
       MOV        AL,*-SP[1]       ;|7|
       CMPB       AL,#5            ;|7|
       B          L3,GT            ;|7|
       ;branch occurs              ;|7|
L2:
       .line  6
       MOV        AL,*,-SP[1]      ;|8|
       ADD        *-SP[2],AL       ;|8|
       .line  5
       INC        *S-P[1]          ;|7|
       MOV        AL,*-SP[1]       ;|7|
       CMPB       AL,#5
       B          L2,LEQ           ;|7|
       ;branch occurs              ;|7|
```

```
L3:
        .line  7
        MOV       AL,*-SP[2]       ;|9|
        .line  8
        SUBB      SP,#2            ;|10|
        LRET
        ;return occurs
        .endfunc     10,000000000h,2
```

Syntax

.member *name*, *value* [, *type*, *storage class*, *size*, *tag*, *dims*]

Description

The **.member** directive defines a member of a structure, union, or enumeration. It is valid only when it appears in a structure, union, or enumeration definition.

- The *name* is the name of the member that is put in the symbol table. The first 128 characters of the name are significant.
- The *value* is the value associated with the member. Any legal expression (absolute or relocatable) is acceptable.
- The *type* is the C type of the member. Appendix A, *Common Object File Format*, contains more information about C types.
- The *storage class* is the C storage class of the member. Appendix A, *Common Object File Format*, contains more information about C storage classes.
- The *size* is the number of bits of memory required to contain this member.
- The *tag* is the name of the type (if any) or structure to which this member belongs. This name *must* have been previously declared by a .stag, .etag, or .utag directive.
- The *dims* is one to four expressions separated by commas; these expressions describe the dimensions of the member.

The order of parameters is significant. The *name* and *value* are required parameters. All other parameters may be omitted or empty. (Adjacent commas indicate an empty entry.) This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example

Following is an example of a C structure definition and the corresponding assembly language statements:

C source:

```
struct doc
{
     char  title;
     char  group;
     int   job_number;
}  doc_info;
```

Resulting assembly language code:

```
FP      .set   AR2
        .file  “ex5.c”
        .stag  _doc,48
        .member _title,0,2,8,16
        .member _group,16,2,8,16
        .member _job_number,32,4,8,16
        .eos
```

Syntax

```
.stag  name  [, size]
     member definitions
.eos
.etag  name  [, size]
     member definitions
.eos
.utag  name  [, size]
     member definitions
.eos
```

Description

The **.stag** directive begins a structure definition. The **.etag** directive begins an enumeration definition. The **.utag** directive begins a union definition. The **.eos** directive ends a structure, enumeration, or union definition.

- The *name* is the name of the structure, enumeration, or union. The first 128 characters of the name are significant. This is a required parameter.
- The *size* is the number of bits that the structure, enumeration, or union occupies in memory. This is an optional parameter; if omitted, the size is unspecified.

The .stag, .etag, or .utag directive is followed by a number of .member directives, which define members in the structure. The .member directive is the only directive that can appear inside a structure, enumeration, or union definition.

The assembler does not allow nested structures, enumerations, or unions. The C compiler unwinds nested structures by defining them separately and then referencing them from the structure in which they are referenced.

Example 1 Following is an example of a structure definition.

C source:

```
struct doc
{
    char title;
    char group;
    int  job_number;
} doc_info;
```

Resulting assembly language code:

```
FP      .set   AR2
        .file  “ex5.c”
        .stag  _doc,48
        .member_title,0,2,8,16
        .member_group,16,2,8,16
        .member_job_number,32,4,8,16
        .eos
```

Example 2 Following is an example of a union definition.

C source:

```
union u_tag {
    int   val1;
    float val2;
    char  valc;
} valu;
```

Resulting assembly language code:

```
        .utag  _u_tag,32
        .member_val1,0,4,11,16
        .member_val2,0,6,11,32
        .member_valc,0,2,11,16
        .eos
```

Example 3 Following is an example of an enumeration definition.

C source:

```
{
    enum o_ty { reg_1, reg_2, result } optypes;
}
```

Resulting assembly language code:

```
        .etag  _o_ty,16
.member_reg_1,0,4,26,26
.member_reg_2,1,4,16,16
.member_result,2,4,16,16
.eos
```

Syntax

.sym *name*, *value* [*, type*, *storage class*, *size*, *tag*, *dims*]

Description

The **.sym** directive specifies symbolic debugging information about a global variable, local variable, or a function.

- The *name* is the name of the variable that is put in the object symbol table. The first 128 characters of the name are significant.
- The *value* is the value associated with the variable. Any legal expression (absolute or relocatable) is acceptable.
- The *type* is the C type of the variable. Appendix A, *Common Object File Format*, contains more information about C types.
- The *storage class* is the C storage class of the variable. Appendix A, *Common Object File Format*, contains more information about C storage classes.
- The *size* is the number of words of memory required to contain this variable.
- The *tag* is the name of the type (if any) or structure to which this variable belongs. This name *must* have been previously declared by a .stag, .etag, or .utag directive.
- The *dims* is one to four expressions separated by commas; these expressions describe the dimensions of the member.

The order of parameters is significant. The *name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example

These lines of C source produce the .sym directives shown below:

C source:

```
struct s { int member1, member2; } str;
int   ext;
int   array[5][10];
long  *ptr;
int   strcmp();

main(arg1,arg2)
   int   arg1;
   char  *arg2;
{
   register r1;
}
```

Resulting assembly language code:

```
FP      .set    AR2
        .global _main
        .global _array
        .bss    _array,50,1,0
        .global _ptr
        .bss    _ptr,1,1,0
        .global _str
        .bss    _str,2,1,0
        .global _ext
        .bss    _ext,1,1,0
        ac27 -q ex6.c ex6.if
        .file   "ex6.c"
        .stag   _s,32
        .member _member1,0,4,8,16
        .member _member2,16,4,8,16
        .eos
        .sect   "text"
        .sym    _main,_main,36,2,0
        .func   7
;***************************************************************
;* FUNCTION NAME: _main                                        *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;* FUNCTION PROPERTIES                                         *
;*                      : 0  Parameter, 2 Auto, 0 SOE          *
;***************************************************************


_main:
;* AL      assigned to _arg1
          .sym   arg1,0,4,17,16
;* AR4     assigned to _arg2
          .sym   arg2,6,18,17,16
          .stm   arg1,-1,4,1,16
          .sym   _arg2,-2,18,1,16
;* AL      assigned to _r1
          .sym   _r1,0,4,4,16
          ADDB    SP,#2
          .line  4
          MOV    *-SP[1],AL       ;|10|
          MOV    *-SP[2],AR4      ;|10|
          .line  6
          SUBB    SP,#2
          LRET
          ;return occurs
          .endfunc            12,000000000h,2

          .sym   _array,_array,244,2,800,5,10
          .sym   _ptr,_ptr,21,2,16
          .sym   _str,_str,8,2,32,_s
          .sym   _ext,_ext,4,2,16
```

Appendix C

Assembler Error Messages

When the assembler completes its second pass, it reports any errors that it encountered during the assembly. It also prints these errors in the listing file (if one is created); an error is printed following the source line that incurred it. You should attempt to correct the first error that occurs in your code before correcting others; a single error condition can cause a cascade of other errors.

If you receive an assembler error message, use this appendix to find solutions to the problems that you encounter. First, locate the error message class number. (The class numbers are listed in alphanumeric order in this appendix.) Then, locate the error message that you received within that class. (Each class number has an alphabetical list of error messages. Each class has a *Description* of the problem and an *Action* that suggests possible remedies.

E0000

Comma required to separate arguments
Comma required to separate parameters
Left parenthesis expected
Left parenthesis is missing
Matching right parenthesis is missing
Missing matching right bracket for condition
Missing right quote of string constant
No matching right parenthesis
Right parenthesis expected
Syntax error
Unrecognized character type
Unrecognized special character

Description These are general syntax errors. The required syntax is not present.

Action Correct the source according to the error message text.

E0002

Illegal mnemonic specified
Invalid mnemonic specification

Description These errors involve invalid mnemonics. The specified instruction, macro, or directive was not recognized.

Action Check the directive or instruction used, then correct the source.

E0003

Cluttered string constant operand encountered
Constant out of range
Illegal conditional operand
Illegal memaddr specification
Illegal register for conditional
Illegal register pair specification
Invalid binary constant specified
Invalid constant specification
Invalid decimal constant specified
Invalid float constant specified
Invalid hex constant specified
Invalid octal constant specified
Memory operand missing offset amount

Description These errors involve invalid operands. The instruction, parameter, or other operand specified was not recognized.

Action Correct the source according to the error message text.

E0004

Absolute, well-defined integer value expected
Cannot use A side register for dest
Conditional not allowed
Identifier expected
Identifier operand expected
IFR illegal as destination register
Illegal character argument specified
Illegal offset mode for 15 bit const
Illegal operand
Illegal register for branch
Illegal string constant operand specified
Illegal structure reference
IN illegal as destination register
Instruction cannot use control register
Invalid data size for relocation
Invalid float constant specified
Invalid identifier, *%s*, specified
Invalid macro parameter specified
Invalid operand, *%c*
Must have one control register
No parameters available for macro arguments
Operand must be register indirect
PC illegal as destination register
Register expected
Single character operand expected
String constant or substitution symbol expected
String operand expected

Structure/Union tag symbol expected
Substitution symbol operand expected

Description These errors involve illegal operands. The instruction, parameter or other operand specified is not legal for this syntax.

Action Correct the source according to the error message text.

E0005

field value operand
Missing operand
Missing operand(s)
Operand missing

Description These errors involve missing operands; a required operand is not supplied.

Action Correct the source so that all required operands are declared.

E0006

.break must occur within a loop
Conditional assembly mismatch
Matching .endloop missing
No matching .endif specified
No matching .endloop specified
No matching .if specified
No matching .loop specified
Open block(s) inside macro
Unmatched .endloop directive
Unmatched .if directive

Description These errors involve unmatched conditional assembly directives. A directive was encountered that requires a matching directive, but the assembler could not find the matching directive.

Action Correct the source according to the error message text.

E0007

Conditional nesting is too deep
Loop count out of range

Description These errors involve conditional assembly loops. Conditional block nesting cannot exceed 32 levels.

Action Correct the .macro/.endmacro, .if/.elseif/.else/.endif, or .loop/.break/.endloop source.

E0008

Bad use of .access directive
Matching .struct directive is not present
Matching .union directive is not present

Description This is an error about unmatched structure definition directives. In a .struct/.endstruct sequence, a directive was encountered that requires a matching directive, but the assembler could not find the matching directive.

Action Check the source for mismatched structure definition directives and correct it.

E0009

B14 or B15 required as long displacement base register
Base address register expected
Base register and index register must be from same file
Base register expected
Can't use relocatable expression in scaled addressing mode
Cannot apply bitwise NOT to floats
Cannot use register offset in unscaled addressing mode
Constant out of range
Illegal struct/union reference dot operator
Matching right bracket is missing
Missing structure/union member or tag
Structure or union tag symbol expected
Structure or union tag symbol not found
Unary operator must be applied to a constant

Description These errors involve an illegally used operator. The operator specified was not legal for the given operands.

Action Correct the source according to the error message text so that all required operands are declared.

E0100

Label missing
Label required
.setsym requires a label

Description These errors involve missing labels. A directive that you used requires a label, but none is specified.

Action Correct the source by specifying the required label for the directive statement.

E0101

Standalone labels not permitted in structure/union defs

Description This error involves an invalid label. Structure and union definitions do not permit a label, but one is specified.

Action Remove the invalid label.

E0102

Local label *%d* defined differently in each pass
Local label *%d* is multiply defined
Local label *%d* is not defined in this section
Local labels can't be used with directives

Description These errors involve the illegal use of local labels.

Action Correct the source according to the error message text. Use .newblock to reuse local labels.

E0200

Bad term in expression
Binary operator can't be applied
Difference between segment symbols not permitted
Divide by zero
Operation cannot be performed on given operands
Unary operator can't be applied
Well–defined expression required

Description These errors occur in general expressions. An illegal operand combination was used, or an arithmetic type is required but not present.

Action Correct the source according to the error message text.

E0201

Absolute operands required for FP operations!
Floating-point divide by zero
Floating-point expression required
Floating-point overflow
Floating-point underflow
Illegal floating-point expression
Invalid floating-point operation

Description These errors occur in floating-point expressions. A floating-point expression was used where an integer expression is required, an integer expression was used where a floating-point expression is required, or a floating-point value is invalid.

Action Correct the source according to the error message text.

E0300

***%s* is not defined in this source file**
***%s* is operand to both .ref and .def**
Can't tag an undefined symbol
Cannot equate an external symbol to an external symbol
Cannot redefine this section name
Empty structure or union definition
Illegal structure or union tag
Missing closing '}' for repeat block
Redefinition of *%s* attempted
Structure tag can't be global
Structure/union member, *%s*, not found
Symbol *%s* has already been defined
Symbol can't be defined in terms of itself
Symbol expected in label field
Symbol expected
The following symbols are undefined:
Union member previously defined
Union tag can't be global

Description These errors involve general symbols. An attempt was made to redefine a symbol or to define a symbol illegally.

Action Correct the source according to the error message text.

E0301

Cannot redefine local substitution symbol
Substitution stack overflow
Substitution symbol not found

Description These errors involve general substitution symbols. An attempt was made to redefine a symbol or to define a symbol illegally.

Action Make sure that the operand of a substitution symbol is defined either as a macro parameter or with a .asg or .eval directive.

E0400

Symbol table entry is not balanced

Description A symbolic debugging directive does not have a complementing directive (for example, a .block without a .endblock).

Action Check the source for mismatched conditional assembly directives, and correct it.

E0500

Macro argument string is too long
Missing macro name
Too many variables declared in macro

Description These errors involve general macros.

Action Correct the source according to the error message text.

E0501

Macro definition not terminated with .endm
Matching .endm missing
Matching .macro missing
.mexit directive outside macro definition
No active macro definition

Description These errors involve macro definition directives. A macro directive does not have a complementing directive (for example, a .macro is used without a .endm).

Action Correct the source according to the error message text.

E0600

***%s* is not in archive format**
***%s* macro library not found**
Bad archive entry for *%s*
Bad archive name
Can't read a line from archive entry
Macro library is not in archive format

Description These errors involve accessing a macro library. A problem was encountered reading from or writing to a macro library archive file. It is likely that the archive file was not created properly.

Action Make sure that the macro libraries are unassembled assembler source files. Also make sure that the macro name and member name are the same and that the extension of the file is .asm.

E0700

Cannot use –g on assembly code with .line directives
Illegal structure/union member
No structure/union currently open
.sym not allowed inside structure/union

Description These errors involve the illegal use of symbolic debugging directives; a symbolic debugging directive is not used in an appropriate place.

Action Correct the source according to the error message text.

E0800

A/B register file mismatch
Cannot perform operation on specified unit
Could not find a valid unit for instruction
Erroneous use of X unit
Illegal destination
Illegal form for LDDW
Illegal functional unit
Illegal memory operand register for unit
Illegal operand combination
Illegal suffix specified for branch
Illegal use of parallel operator
Instruction cannot use X unit
Instructions not permitted in structure/union definitions
Offset too large
Unit specifier disagrees with operation

Description These errors involve illegal operands. The instruction, parameter, or other operand specified was not legal for this syntax.

Action Correct the source according to the error message text.

E0801

Processor resource allocation conflict
Too many branches to labels in this packet
Too many multi–cycle NOPs in this packet
Too many reads from one register in this packet

Description

Action Correct the source according to the error message text.

E0900

Can’t include a file inside a loop or macro
Cannot change version after 1st instruction
Illegal structure definition contents
Illegal structure member
Illegal union definition contents
Illegal union member
Invalid load-time label
Invalid structure/union contents
.var allowed only within macro definitions

Description These errors involve illegally used directives. Directives were encountered where they are not permitted. (The directives are not permitted in that position because they would cause a corruption of the object file.) Many directives are not permitted inside of structure or union definitions.

Action Correct the source according to the error message text.

E1000

Include/Copy file not found or opened

Description The specified filename cannot be found.

Action Check spelling, pathname, environment variables, etc.

E1300

Copy limit has been reached
Exceeded limit for macro arguments
Macro nesting limit exceeded

Description These errors involve general assembler limits that have been exceeded. The nesting of .copy/.include files is limited to ten levels. Macro arguments are limited to 32 parameters. Macro nesting is limited to 32 levels.

Action Check the source to determine how limits have been exceeded.

E9999

***%s* defined differently in each pass**

Description A symbol in the symbol table did not have the same value in pass1 and pass2. You likely have an error in a directive, macro, or label.

Action Check the source to determine what caused the problem and correct the source.

E9999

Can't push *%s* on expr stack
Pass conflict

Description These are internal assembler errors. If they occur repeatedly, the assembler may be corrupt or confused.

Action Try to assemble a smaller file. If a smaller file does not assemble, reinstall the assembler.

W0000

Delay slot count must be 1 to 9, 1 assumed
Half–word offsets must be divisible by 2, truncated
Invalid page number specified – ignored
No operands expected. Operands ignored
Specified alignment is outside accessible memory – ignored
Too many operands
Trailing Operands Ignored
Word offsets must be divisible by 4, truncated

Description These are warnings about operands. The assembler encountered operands that it did not expect.

Action Check the source to determine what caused the problem and whether you need to correct the source.

W0001

Field value truncated to *%ld*
Field width truncated to *%d*
Maximum alignment is to 32K boundary—alignment ignored
Power of 2 required, *%ld* assumed
Section Name is limited to 8 characters
Section name, *%s*, truncated to 8 characters
String is too long—will be truncated
Value truncated
Value truncated to *%d*-bit width
Value truncated to byte size

Description These warnings involve truncated values. The expression given was too large to fit within the instruction opcode or the required number of bits.

Action Check the source to make sure the result is acceptable or change the source if an error has occurred.

W0002

Address expression will wrap-around
Expression will overflow, value truncated

Description These are warnings about arithmetic expressions. The assembler has performed a calculation that will produce the indicated result, which may or may not be acceptable.

Action Determine whether the result is acceptable; change the source if an error has occurred.

W0003

.sym for *function name* required before .func

Description This is a warning about problems with symbolic debugging directives. A .sym directive defining the function does not appear before the .func directive.

Action Correct the source according to the error message text.

W0004

.access only allowed in top–most structure definition
Access point has already been defined
Illegal unit specifier, ignored
Open block(s) at EOF

Description These are warnings about problems with structure definitions.

Action Correct the source according to the error message text.

W9999

Assembler error not defined

Description This warning is produced when the error does not fall into any other error category.

Action Check the source to determine what caused the error and whether you need to correct the source.

Appendix D

Linker Error Messages

This appendix lists the linker error messages in alphabetical order. In these listings, the symbol (...) represents the name of an object that the linker is attempting to interact with when an error occurs.

A

absolute symbol (...) being redefined

Description An absolute symbol cannot be redefined.

Action Check the syntax of all expressions, and check the input directives for accuracy.

adding name (...) to multiple output sections

Description An input section is mentioned more than once in the SECTIONS directive.

Action Modify the SECTIONS directive in your linker command file.

ALIGN illegal in this context

Description Alignment of a symbol is performed outside of a SECTIONS directive.

Action Modify your linker command file and move the align specification inside the SECTIONS directive.

alignment for (...) must be a power of 2

Description Section alignment was not specified as a power of 2.

Action Make sure that in hexadecimal, all powers of 2 consist of the integers 1, 2, 4, or 8 followed by a series of zero or more 0s.

alignment for (...) redefined

Description More than one alignment is supplied for a section.

Action Modify your linker command file.

attempt to decrement DOT

Description A statement such as .–= value is supplied; this is illegal. Assignments to the . symbol can be used only to create holes.

Action Modify your linker command file.

B

bad fill value

Description The fill value must be a 16-bit constant.

Action Modify the fill specifications in your linker command file.

binding address (...) for section (...) is outside all memory on page (...)

Description Each section must fall within memory configured with the MEMORY directive.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are placed in unconfigured memory.

binding address (...) for section (...) overlays (...) at (...)

Description Two sections overlap and cannot be allocated.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are placed in unconfigured memory.

binding address (...) incompatible with alignment for section (...)

Description The section has an alignment requirement from an .align directive or previous link. The binding address violates this requirement.

Action Modify your linker command file.

binding address for (...) redefined

Description More than one binding value is supplied for a section.

Action Modify your linker command file and remove all binding values except one.

blocking for (...) must be a power of 2

Description Section blocking is not a power of 2.

Action Make sure that in hexadecimal, all powers of 2 consist of the integers 1, 2, 4, or 8 followed by a series of zero or more 0s.

blocking for (...) redefined

Description More than one blocking value is supplied for a section.

Action Modify your linker command file, and remove all blocking values except one.

C

–c requires fill value of 0 in .cinit (... overridden)

Description The .cinit tables must be terminated with 0; therefore, the fill value of the .cinit section must be 0.

Action Modify your linker command file.

cannot complete output file (...), write error

Description This usually means that the file system is out of space.

Action Check the disk volume; delete files or add more disk space.

cannot create output file (...)

Description This usually indicates an illegal filename.

Action Check the spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

cannot resize (...), section has initialized definition in (...)

Description An *initialized* input section named .stack or .heap exists, preventing the linker from resizing the section.

Action Modify your linker command file.

cannot specify a page for a section within a GROUP

Description A section was specified to a particular page within a group. The entire group is treated as one unit, so the group can be specified to a page of memory, but the sections making up the group cannot be specified individually.

Action Modify your linker command file.

cannot specify both binding and memory area for (...)

Description Both binding and named memory were specified. The two are mutually exclusive.

Action If you want the code to be placed at a specific address, use binding only. If you want the code to be placed into a range defined in the MEMORY directive, use named memory only.

can’t align a section within GROUP – (...) not aligned

Description A section in a group was specified for individual alignment. The entire group is treated as one unit, so the group can be aligned or bound to an address, but the sections making up the group cannot be aligned individually.

Action Modify your linker command file.

can’t align within UNION – section (...) not aligned

Description A section in a union was specified for individual alignment. The entire union is treated as one unit, so the union can be aligned or bound to an address, but the sections making up the union cannot be handled individually.

Action Modify your linker command file.

can’t allocate (...), size ... (page ...)

Description A section cannot be allocated, because no existing configured memory area is large enough to hold it.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are placed in unconfigured memory.

can’t create map file (...)

Description Usually indicates an illegal filename.

Action Check the spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t find input file *filename*

Description The file, *filename*, is not in your PATH, is misspelled, etc.

Action Check spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t open (...)

Description The specified file does not exist.

Action Check the spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t open *filename*

Description The file with the specified filename cannot be opened; for some reason; the file does not exist, or the wrong file type is incorrect, etc.

Action Check the spelling, pathname, environment variables, etc.

can’t read (...)

Description The file may be corrupt.

Action Try reassembling the input file.

can’t seek (...)

Description The file may be corrupt.

Action Try reassembling the input file.

can’t write (...)

Description Disk may be full or protected.

Action Check the disk volume and protection; ensure that the disk is not write protected or create space as needed.

command file nesting exceeded with file (...)

Description Command file nesting is allowed up to 16 levels.

Action Modify your linker command file.

E

–e flag does not specify a legal symbol name (...)

Description The –e option is not supplied with a valid symbol name as an operand.

Action Use a valid symbol name with the –e option.

entry point other than _c_int00 specified

Description This error message occurs only if you use file –c or –cr option invoking the linker. A program entry point other than the value of _c_int00 was supplied. The runtime conventions of the compiler assume that _c_int00 is the only entry point.

Action No action is required. To avoid this warning, do not redefine the program entry point at the same time that you use the –c or –cr option.

entry point symbol (...) undefined

Description The symbol used with the –e option is not defined.

Action Be sure that the symbol name that you use with the –e option is defined.

errors in input – (...) not built

Description Previous linker errors prevent the creation of an output file.

Action Correct the other errors that the linker lists, then relink the files.

F

fail to copy (...)

Description The file may be corrupt.

Action Try reassembling the input file.

fail to read (...)

Description The file may be corrupt.

Action Try reassembling the input file.

fail to seek (...)

Description The file may be corrupt.

Action Try reassembling the input file.

fail to skip (...)

Description The file may be corrupt.

Action Try reassembling the input file.

fail to write (...)

Description The disk may be full or protected.

Action Check disk volume and protection; ensure that the disk is not write protected, or create space as needed.

file (...) has no relocation information

Description You have attempted to relink a file that was not linked with –r.

Action Use the –r linker option to link all files that you plan to relink; this retains the necessary relocation information.

file (...) is of unknown type, magic number = (...)

Description The binary input file is not a COFF file.

Action Ensure that all input files to the linker are in the TMS320C27x™ COFF format.

fill value for (...) redefined

Description More than one fill value is supplied for an output section. Individual holes cannot be filled with different values with the section definition.

Action Modify your linker command file.

I

–i path too long (...)

Description The number of characters in an –i path exceeds the maximum limit of 256.

Action Use a pathname that is 256 characters or less.

illegal input character

Description There is a control character or other unrecognized character in the command file.

Action Modify your linker command file.

illegal memory attributes for (...)

Description The attributes are not some combination of R, W, I, and X.

Action Modify your linker command file.

illegal operator in expression

Description The linker detected an illegal expression operator.

Action Review legal expression operators shown in Table 7–2 on page 7-57, and modify your code accordingly.

illegal option within SECTIONS

Description An invalid option was used with the SECTIONS directive.

Action Use only the –l (lowercase L) option with a SECTIONS directive.

illegal relocation type (...) found in section(s) of file (...)

Description The binary file is corrupt.

Action Inspect the object file(s), and rebuild the file(s) as necessary.

internal error (...)

Description This linker has an internal error.

Action Contact the microcontroller hotline.

invalid archive size for file (...)

Description The archive file is corrupt.

Action Inspect the archive file, and rebuild it as necessary.

invalid path specified with –i flag

Description The operand of the –i option (flag) is not a valid pathname.

Action Ensure that the pathname you use with the –i option is valid.

invalid value for –f flag

Description The value for –f option (flag) is not a 4-byte (32-bit) constant.

Action Use a 4-byte constant with the –f option.

invalid value for –heap flag

Description The value for –heap option (flag) is not a 4-byte (32-bit) constant.

Action Use a 4-byte constant with the –heap option.

invalid value for –stack flag

Description The value for –stack option (flag) is not a 4-byte (32-bit) constant.

Action Use a 4-byte constant with the –stack option.

invalid value for –v flag

Description The value for –v option (flag) is not a constant.

Action Use a constant with the –v option.

I/O error on output file (...)

Description The disk may be full or protected.

Action Check the disk volume and protection; ensure that the disk is not write protected or create space as needed.

L

length redefined for memory area (...)

Description A memory area in a MEMORY directive has more than one length.

Action Modify your linker command file.

library (...) member (...) has no relocation information

Description The library member named in the message has no relocation information, which means that it cannot satisfy unresolved references in other files when linking.

Action This warning requires no action. The library member serves no purpose since it has no relocation information, and the linker ignores it.

line number entry found for absolute symbol

Description The input file may be corrupt.

Action Try reassembling the input file.

load address for uninitialized section (...) ignored

Description A load address is supplied for an uninitialized section. Uninitialized sections have no load addresses, only run addresses.

Action Modify your linker command file and remove the load address specification for the uninitialized section.

load address for UNION ignored

Description UNION refers only to the section's run address.

Action Modify your linker command file.

load allocation required for initialized UNION member (...)

Description A load address is supplied for an initialized section in a union. UNIONs refer to runtime allocation only.

Action Specify the load address for all sections within a union separately. Modify your linker command file accordingly.

M

–m flag does not specify a valid filename

Description You did not specify a valid filename for the file to which you are writing the output map.

Action Ensure that the filename you use with the –m option is a valid filename.

making aux entry *filename* for symbol *n* out of sequence

Description The input file may be corrupt.

Action Try reassembling the input file.

memory area for (...) redefined

Description More than one named memory allocation is supplied for an output section.

Action Modify your linker command file.

memory attributes redefined for (...)

Description More than one set of memory attributes is supplied for an output section.

Action Modify your linker command file.

memory page for (...) redefined

Description More than one page allocation is supplied for a section.

Action Modify your linker command file.

memory types (...) and (...) on page (...) overlap

Description Memory ranges on the same page overlap.

Action If you are using a linker command file, ensure that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are placed in unconfigured memory.

missing filename on –l; use –l <filename>

Description No filename operand is supplied for the –l (lowercase L) option.

Action Specify a filename with the –l option to name a library that is not in the current directory.

misuse of DOT symbol in assignment instruction

Description The . symbol is used in an assignment statement that is outside the SECTIONS directive.

Action Modify your linker command file.

N

no allocation allowed for uninitialized UNION member

Description A load address was supplied for an uninitialized section in a union. An uninitialized section in a union gets its run address from the UNION statement and has no load address, so no load allocation is valid for the member.

Action Modify your linker command file.

no allocation allowed with a GROUP–allocation for section (...) ignored

Description A section in a group was specified for individual allocation. The entire group is treated as one unit, so the group can be aligned or bound to an address, but the sections making up the group cannot be handled individually.

Action Modify your linker command file and remove that allocation specification.

no input files

Description No COFF files were supplied. The linker cannot operate without at least one input COFF file.

Action Name at least one COFF file as input when you invoke the linker.

no load address specified for (...); using run address

Description No load address is supplied for an initialized section. If an initialized section has a run address only, the section is allocated to run and load at the same address.

Action No action is required. The linker automatically assumes that you want the the load address to be the same as the run address.

no run allocation allowed for UNION member (...)

Description A UNION defines the run address for all of its members; therefore, individual run allocations are illegal.

Action Modify your linker command file.

no string table in file *filename*

Description The input file may be corrupt.

Action Try reassembling the input file.

no symbol map produced – not enough memory

Description Available memory is insufficient to produce the symbol list. This is a nonfatal condition that prevents the generation of the symbol list in the map file.

Action Increase the available memory in your system.

O

–o flag does not specify a valid file name : (...)

Description The filename used with the –o option does not follow the operating-system file naming conventions.

Action Be sure the filename that you specify with the –o option follows the operating-system file naming conventions.

origin missing for memory area (...)

Description An origin is not specified with the MEMORY directive.

Action Modify your linker command file and include an origin value in the MEMORY directive to specify the starting address of a memory range.

out of memory, aborting

Description Your system does not have enough memory to perform all required tasks.

Action Try breaking the assembly language files into multiple smaller files and partially link them. See section 7.16, *Partial (Incremental) Linking*, page 7-63.

output file has no .bss section

Description This is a warning. The .bss section is usually present in a COFF file, but there is no real requirement for it to be present.

Action To avoid this warning, specify the .bss section in your linker command file.

output file has no .data section

Description This is a warning. The .data section is usually present in a COFF file, but there is no real requirement for it to be present.

Action To avoid this warning, specify the .data section in your linker command file.

output file has no .text section

Description This is a warning. The .text section is usually present in a COFF file, but there is no real requirement for it to be present.

Action To avoid this warning, specify the .text section in your linker command file.

output file (...) not executable

Description The output file may have unresolved symbols or problems stemming from other errors. This condition is not fatal.

Action No action is required. This warning tells you that your code will not be fully linked .

overwriting aux entry *filename* of symbol *n*

Description The input file may be corrupt.

Action Try reassembling the input file.

P

PC-relative displacement overflow at address (...) in file (...)

Description The relocation of a PC-relative operand resulted in a displacement too large to encode in the instruction.

Action Modify the memory map so that displacements are within range.

R

–r incompatible with –s (–s ignored)

Description Both the –r option and the –s option were used. Since the –s option strips the relocation information and –r requests a relocatable object file, these options are in conflict with each other.

Action To avoid this warning, do not use the –s option with the –r option. If you use these options together, the –s option is ignored.

relocation entries out of order in section (...) of file (...)

Description The input file may be corrupt.

Action Try reassembling the input file.

relocation symbol not found: index (...), section (...), file (...)

Description The input file may be corrupt.

Action Try reassembling the input file.

S

section (...) at (...) overlays at address (...)

Description Two sections overlap and cannot be allocated.

Action If you are using a linker command file, ensure that MEMORY and SECTIONS directives allow enough room to ensure that no sections overlap.

section (...) enters unconfigured memory at address (...)

Description A section cannot be allocated because no existing configured memory area is large enough to hold it.

Action If you are using a linker command file, ensure that MEMORY and SECTIONS directives allow enough room to ensure that no sections are placed in unconfigured memory.

section (...) not built

Description There is a syntax error in the SECTIONS directive.

Action Inspect and modify the SECTIONS directive defined in your linker command file.

section (...) not found

Description An input section specified in a SECTIONS directive was not found in the input file.

Action Modify your linker command file and ensure that the input section specified exists in one of the input files.

section (...) won't fit into configured memory

Description A section cannot be allocated, because no configured memory area is large enough to hold it.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are placed in unconfigured memory.

seek to (...) failed

Description The input file may be corrupt.

Action Try reassembling the input file.

semicolon required after assignment

Description There is a syntax error in the command file.

Action Modify your linker command file.

statement ignored

Description There is a syntax error in an expression.

Action Modify your linker command file.

symbol (...) from file (...) being redefined

Description A defined symbol is redefined in an assignment statement.

Action No action is required. To avoid this warning, remove one of the symbol definitions in the linker command file.

symbol referencing errors — (...) not built

Description Symbol references could not be resolved. Therefore, an object module could not be built.

Action Be sure that all references are satisfied by the input files in order to build an executable.

T

too many arguments – use a command file

Description You used too many arguments on a command line or in response to prompts.

Action Create a linker command file to name all of the arguments that you want to pass to the linker.

too many –i options, 7 allowed

Description More than seven –i options were used.

Action Use the C_DIR or A_DIR environment variable to name additional search directories.

type flags for (...) redefined

Description More than one section type is supplied for a section. Note that type COPY has all of the attributes of type DSECT, so DSECT does not need to be specified separately.

Action Modify your linker command file.

type flags not allowed for GROUP or UNION

Description A type is specified for a section in a group or union. Special section types apply to individual sections only.

Action Modify your linker command file, and supply only one section type for a section.

U

–u does not specify a legal symbol name

Description You did not specify a symbol name with the –u option.

Action Specify a valid symbol name with the –u option.

undefined symbol (...) first referenced in file (...)

Description Either a referenced symbol is not defined, or the –r option was not used. Unless the –r option is used, the linker requires that all referenced symbols be defined. This condition prevents the creation of an executable output file.

Action Link using the –r option, or define the symbol.

undefined symbol in expression

Description An assignment statement contains an undefined symbol.

Action Modify your linker command file.

unexpected EOF(end of file)

Description There is a syntax error in the linker command file.

Action Modify your linker command file.

unrecognized option (...)

Description You tried to use an option that the linker did not recognize.

Action Check the list of valid options. See Table 7–1 on page 7-7.

Z

zero or missing length for memory area (...)

Description A memory range defined with the MEMORY directive did not have a nonzero length.

Action Modify your linker command file.

Appendix E

Glossary

A

absolute address: An address that is permanently assigned to a TMS320C27x™ memory location.

alignment: A process in which the linker places an output section at an address that falls on an *n*-byte boundary, where *n* is a power of 2. You can specify alignment with the SECTIONS linker directive.

allocation: A process in which the linker calculates the final memory addresses of output sections.

archive library: A collection of individual files that have been grouped into a single file.

archiver: A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

ASCII: American Standard Code for Information 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 directives. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

assembly-time constant: A symbol that is assigned a constant value with the .set directive.

assignment statement: A statement that assigns a value to a variable.

autoinitialization: The process of initializing global C variables (contained in the .cinit section) before beginning program execution.

auxiliary entry: The extra entry that a symbol may have in the symbol table that contains additional information about the symbol (whether it is a filename, a section name, a function name, etc.).

B

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 declarations and statements that are grouped together with braces.

.bss: One of the default COFF sections. You can use the .bss directive to reserve a specified amount of space in the memory map that can later be used for storing data. The .bss section is uninitialized.

byte: A sequence of eight adjacent bits operated upon as a unit.

C

C/C++ compiler: A program that translates C/C++ source statements into assembly language source statements.

command file: A file that contains options, filenames, directives, or commands for the linker or hex-conversion utility.

comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

common object file format (COFF): A binary object file format that promotes modular programming by supporting the concept of *sections*. All COFF sections are independently relocatable in memory space; you can place any section into any allocated block of target memory.

conditional processing: A method of processing one block of source code or an alternate block of source code according to the evaluation of a specified expression.

configured memory: Memory that the linker has specified for allocation.

constant: A numeric value that does not change and that can be used as an operand.

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, the line they were defined on, the lines that referenced them, and their final values.

D

.data: One of the default COFF sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

directives: Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

E

emulator: A hardware development system that emulates TMS320C2700 operation.

entry point: The starting execution point in target memory.

executable module: An object file that has been linked and can be executed in a TMS320C27x system.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

external symbol: A symbol that is used in the current program module but is defined in a different program module.

F

field: For the TMS320C27x, a software-configurable data type whose length can be programmed to be any value in the range of 1–32 bits.

file header: A portion of a COFF object file that contains general information about the object file, such as the number of section headers, the type of system the object file can be downloaded to, the number of symbols in the symbol table, and the symbol table's starting address.

G

global symbol: A symbol that is either 1) defined in the current module and accessed in another module, or 2) accessed in the current module but defined in another module.

GROUP: An option of the SECTIONS directive that forces specified output sections to be allocated contiguously (as a group).

H

hex-conversion utility: A program that accepts COFF files and converts them into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer.

high-level language debugging: The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

hole: An area containing no actual code or data. This area is between the input sections that compose an output section.

I

incremental linking: Linking files in several passes. Incremental linking is useful for large applications, because you can partition the application, link the parts separately, and then link all of the parts together.

initialized section: A COFF section that contains executable code or initialized data. An initialized section can be built up with the .data, .text, or .sect directive.

input section: A section from an object file that is linked into an executable module.

L

label: A symbol that begins in column 1 of a source statement and corresponds to the address of that statement.

line-number entry: An entry in a COFF output module that maps lines of assembly code back to the original C source file that created them.

linker: A software tool that combines object files to form an object module that can be allocated into TMS320C27x system memory and executed by the device.

listing file: An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the SPC.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

loader: A device that loads an executable module into TMS320C27x system memory.

M

macro: A user-defined routine that can be used as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro.

macro expansion: The source statements that are substituted for the macro call and subsequently assembled.

macro library: An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of .asm.

magic number: A COFF file header entry that identifies an object file as a module that can be executed by the TMS320C27x.

map file: An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which they were defined.

member: The elements or variables of a structure, union, archive, or enumeration.

memory map: A map of target system memory space, which is partitioned into functional blocks.

mnemonic: An instruction name that the assembler translates into machine code.

model statement: Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.

N

named section: An initialized section that is defined with a .sect directive.

O

object file: A file that has been assembled or linked and that contains machine-language object code.

object library: An archive library made up of individual object files.

operands: The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.

optional header: A portion of a COFF object file that the linker uses to perform relocation at download time.

options: Command parameters that allow you to request additional or specific functions when you invoke a software tool.

output module: A linked, executable object file that can be downloaded and executed on a target system.

output section: A final, allocated section in a linked, executable module.

P

partial linking: Linking files in several passes. Incremental linking is useful for large applications because you can partition the application, link the parts separately, and then link all of the parts together.

Q

quiet run: An option that suppresses the normal banner and the progress information.

R

raw data: Executable code or initialized data in an output section.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

run address: The address where a section runs.

S

section: A relocatable block of code or data that ultimately occupies contiguous space in the TMS320C27x memory map.

section header: A portion of a COFF object file that contains information about a section in the file. Each section has its own header; the header points to the section's starting address, contains the section's size, etc.

section program counter: See *SPC*

sign extend: To fill the unused MSBs of a value with the value's sign bit.

simulator: A software development system that simulates TMS320C27x operation.

source file: A file that contains C code or assembly language code that is compiled or assembled to form an object file.

SPC (section program counter): An element that keeps track of the current location within a section; each section has its own SPC.

static variable: An element whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; the previous value is resumed when the function or program is reentered.

storage class: Any entry in the symbol table that indicates how a symbol is accessed.

string table: A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead, they are stored in the string table). The name portion of the symbol's entry points to the location of the string in the string table.

structure: A collection of one or more variables grouped together under a single name.

subsection: A relocatable block of code or data that will ultimately occupy continuous space in the TMS320C27x memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.

symbol: A string of alphanumeric characters that represents an address or a value.

symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool, such as a simulator or an emulator.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

T

tag: An optional *type* name that can be assigned to a structure, union, or enumeration.

target memory: Physical memory in a TMS320C27x system into which executable object code is loaded.

.text: One of the default COFF sections. The .text section is an initialized section that contains executable code. You can use the .text directive to assemble code into the .text section.

U

unconfigured memory: Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section: A COFF section that reserves space in the memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

UNION: An option of the SECTIONS directive that causes the linker to allocate the same address to multiple sections.

union: A variable that can hold objects of different types and sizes.

unsigned value: An element that is treated as a positive number, regardless of its actual sign.

W

well-defined expression: A term or group of terms that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

word: A 16-bit addressable location in target memory.

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X