# TMS320C28x Assembly Language Tools User's Guide

Literature Number: SPRU513B
March 2006

TEXAS INSTRUMENTS

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

| **Products** | | **Applications** | |
|---|---|---|---|
| Amplifiers | amplifier.ti.com | Audio | www.ti.com/audio |
| Data Converters | dataconverter.ti.com | Automotive | www.ti.com/automotive |
| DSP | dsp.ti.com | Broadband | www.ti.com/broadband |
| Interface | interface.ti.com | Digital Control | www.ti.com/digitalcontrol |
| Logic | logic.ti.com | Military | www.ti.com/military |
| Power Mgmt | power.ti.com | Optical Networking | www.ti.com/opticalnetwork |
| Microcontrollers | microcontroller.ti.com | Security | www.ti.com/security |
| | | Telephony | www.ti.com/telephony |
| | | Video & Imaging | www.ti.com/video |
| | | Wireless | www.ti.com/wireless |

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

# Read This First

## *About This Manual*

The *TMS320C28x Assembly Language Tools User's Guide* tells you how to use these assembly language tools:

- ❑ Assembler
- ❑ Archiver
- ❑ Linker
- ❑ Absolute lister
- ❑ Cross-reference lister
- ❑ Object file display utility
- ❑ Name utility
- ❑ Strip utility
- ❑ 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 TMS320C28x 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 TMS320C28x 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 11, 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 D, 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 TMS320C28x 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 TMS320C28x core is also referred to as TMS320C28x or C28x.

- 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   XAR1 , #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:

  **cl2000 −v28 −z** [*options*] *filename*$_{1,}$ ...[ *filename*$_n$]

  The **cl2000 −v28 −z** 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   value1 [, ... , valuen]
  ```

  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 TMS320C28x 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):

***TMS320C28x Optimizing C/C++ Compiler User's Guide*** (literature number SPRU514) describes the TMS320C28x C/C++ compiler. This C/C++ compiler accepts ANSI standard C/C++ source code and produces TMS320 assembly language source code for the TMS320C28x device.

***Code Composer Studio User's Guide*** (literature number SPRU328) explains how to use the Code Composer development environment to build and debug embedded real-time DSP applications.

***TMS320C28x DSP CPU and Instruction Set Reference Guide*** (literature number SPRU430) describes the central processing unit (CPU) and the assembly language instructions of the TMS320C28x™ fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

***TMS320C2xx User's Guide*** (literature number SPRU127) discusses the hardware aspects of the TMS320C2xx™ 16-bit fixed-point digital signal processors. It describes the architecture, the instruction set, and the on-chip peripherals.

## 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

# Contents

**1 Introduction to the Software Development Tools . . . . . . . . 1-1**

*Provides an overview of the software development tools.*

1.1 Software Development Tools Overview . . . . . . . . 1-2
1.2 Tools Descriptions . . . . . . . . 1-3

**2 Introduction to Common Object File Format . . . . . . . . 2-1**

*Discusses the basic COFF concept of sections and how they can help you use the assembler and linker more efficiently. Read this chapter before using the assembler and linker.*

2.1 Sections . . . . . . . . 2-2
2.2 How the Assembler Handles Sections . . . . . . . . 2-4
2.2.1 Uninitialized Sections . . . . . . . . 2-4
2.2.2 Initialized Sections . . . . . . . . 2-5
2.2.3 Named Sections . . . . . . . . 2-6
2.2.4 Subsections . . . . . . . . 2-7
2.2.5 Section Program Counters . . . . . . . . 2-8
2.2.6 An Example That Uses Sections Directives . . . . . . . . 2-8
2.3 How the Linker Handles Sections . . . . . . . . 2-11
2.3.1 Default Memory Allocation . . . . . . . . 2-12
2.3.2 Placing Sections in the Memory Map . . . . . . . . 2-13
2.4 Relocation . . . . . . . . 2-14
2.4.1 Run-Time Relocation . . . . . . . . 2-16
2.5 Loading a Program . . . . . . . . 2-17
2.6 Symbols in a COFF File . . . . . . . . 2-18
2.6.1 External Symbols . . . . . . . . 2-18
2.6.2 The Symbol Table . . . . . . . . 2-19

**3 Assembler Description . . . . . . . . 3-1**

*Explains how to invoke the assembler and discusses source statement format, valid constants and expressions, and assembler output.*

3.1 Assembler Overview . . . . . . . . 3-2
3.2 The Assembler’s Role in the Software Development Flow . . . . . . . . 3-3
3.3 Invoking the Assembler . . . . . . . . 3-4
3.4 Naming Alternate Directories for Assembler Input . . . . . . . . 3-7
3.4.1 Using the −I Assembler Option . . . . . . . . 3-7
3.4.2 Using the C2000_A_DIR and A_DIR Environment Variables . . . . . . . . 3-8

3.5 Source Statement Format . . . . . 3-10
  3.5.1 Label Field . . . . . 3-11
  3.5.2 Mnemonic Field . . . . . 3-12
  3.5.3 Operand Field . . . . . 3-12
  3.5.4 Comment Field . . . . . 3-12
3.6 Constants . . . . . 3-13
  3.6.1 Binary Integers . . . . . 3-13
  3.6.2 Octal Integers . . . . . 3-14
  3.6.3 Decimal Integers . . . . . 3-14
  3.6.4 Hexadecimal Integers . . . . . 3-15
  3.6.5 Character Constants . . . . . 3-15
  3.6.6 Assembly-Time Constants . . . . . 3-16
  3.6.7 Floating-Point Constants . . . . . 3-16
3.7 Character Strings . . . . . 3-17
3.8 Symbols . . . . . 3-18
  3.8.1 Labels . . . . . 3-18
  3.8.2 Local Labels . . . . . 3-18
  3.8.3 Symbolic Constants . . . . . 3-21
  3.8.4 Defining Symbolic Constants (−d Option) . . . . . 3-21
  3.8.5 Predefined Symbolic Constants . . . . . 3-22
  3.8.6 Substitution Symbols . . . . . 3-24
3.9 Expressions . . . . . 3-25
  3.9.1 Operators . . . . . 3-25
  3.9.2 Expression Overflow and Underflow . . . . . 3-26
  3.9.3 Well-Defined Expressions . . . . . 3-27
  3.9.4 Conditional Expressions . . . . . 3-27
  3.9.5 Legal Expressions . . . . . 3-27
3.10 Built-In Functions . . . . . 3-29
3.11 Source Listings . . . . . 3-31
3.12 Debugging Assembly Source . . . . . 3-34
3.13 Cross-Reference Listings . . . . . 3-36
3.14 Smart Encoding . . . . . 3-37
3.15 C-Type Symbolic Debugging for Assembly Variables (−mg option) . . . . . 3-39
3.16 TMS320C28x Assembler Modes . . . . . 3-41
  3.16.1 C27x Object Mode . . . . . 3-41
  3.16.2 C28x Object Mode . . . . . 3-42
  3.16.3 C28x Object − Accept C27x Syntax Mode . . . . . 3-43
  3.16.4 C28x Object − Accept C2xlp Syntax Mode . . . . . 3-43

**4 Assembler Directives . . . . . 4-1**

*Describes the directives according to function and presents the directives in alphabetical order.*

4.1 Directives Summary . . . . . 4-2
4.2 Compatibility With the TMS320C1x/C2x/C2xx/C5x Assembler Directives . . . . . 4-7

4.3 Directives That Define Sections . . . . 4-8
4.4 Directives That Initialize Constants . . . . 4-10
4.5 Directive That Aligns the Section Program Counter . . . . 4-13
4.6 Directives That Format the Output Listing . . . . 4-15
4.7 Directives That Reference Other Files . . . . 4-17
4.8 Directives That Enable Conditional Assembly . . . . 4-18
4.9 Directives That Define Symbols at Assembly Time . . . . 4-19
4.10 Assembler Mode Override Directives . . . . 4-20
4.11 Miscellaneous Directives . . . . 4-21
4.12 Directives Reference . . . . 4-22

**5 Macro Language . . . . 5-1**

*Describes macro directives, substitution symbols used as macro parameters, and how to create macros.*

5.1 Using Macros . . . . 5-2
5.2 Defining Macros . . . . 5-3
5.3 Macro Parameters/Substitution Symbols . . . . 5-5
5.3.1 Directives That Define Substitution Symbols . . . . 5-6
5.3.2 Built-In Substitution Symbol Functions . . . . 5-8
5.3.3 Recursive Substitution Symbols . . . . 5-9
5.3.4 Forced Substitutions . . . . 5-10
5.3.5 Accessing Individual Characters of Subscripted Substitution Symbols . . . . 5-11
5.3.6 Substitution Symbols as Local Variables in Macros . . . . 5-12
5.4 Macro Libraries . . . . 5-13
5.5 Using Conditional Assembly in Macros . . . . 5-14
5.6 Using Labels in Macros . . . . 5-16
5.7 Producing Messages in Macros . . . . 5-17
5.8 Using Directives to Format the Output Listing . . . . 5-19
5.9 Using Recursive and Nested Macros . . . . 5-21
5.10 Macro Directives Summary . . . . 5-23

**6 Archiver Description . . . . 6-1**

*Describes instructions for invoking the archiver, creating new archive libraries, and modifying existing libraries.*

6.1 Archiver Overview . . . . 6-2
6.2 Archiver Development Flow . . . . 6-3
6.3 Invoking the Archiver . . . . 6-4
6.4 Archiver Examples . . . . 6-6

**7 Linker Description . . . . 7-1**

*Explains how to invoke the linker, provides details about linker operation, discusses linker directives, and presents a detailed linking example.*

7.1 Linker Overview . . . . 7-2
7.2 The Linker’s Role in the Software Development Flow . . . . 7-3

7.3 Invoking the Linker . . . . . . . . . . 7-4
7.4 Linker Options . . . . . . . . . . 7-5
7.4.1 Relocation Capabilities (−a and −r Options) . . . . . . . . . . 7-6
7.4.2 Disable Merge of Symbolic Debugging Information (−b Option) . . . . . . . . . . 7-8
7.4.3 C Language Options (−c and −cr Options) . . . . . . . . . . 7-8
7.4.4 Define an Entry Point (−e Option) . . . . . . . . . . 7-9
7.4.5 Set Default Fill Value (−f fill_value Option) . . . . . . . . . . 7-9
7.4.6 Make a Symbol Global (−g symbol Option) . . . . . . . . . . 7-9
7.4.7 Make All Global Symbols Static (−h Option) . . . . . . . . . . 7-10
7.4.8 Define Heap Size (−heap size Option) . . . . . . . . . . 7-10
7.4.9 Alter the Library Search Algorithm (−l Option, −I Option, and C_DIR Environment Variable) . . . . . . . . . . 7-10
7.4.10 Disable Conditional Linking (−j Option) . . . . . . . . . . 7-13
7.4.11 Ignore Alignment (−k Option) . . . . . . . . . . 7-13
7.4.12 Create a Map File (−m filename Option) . . . . . . . . . . 7-13
7.4.13 Name an Output Module (−o filename Option) . . . . . . . . . . 7-14
7.4.14 Specify a Quiet Run (−q Option) . . . . . . . . . . 7-15
7.4.15 Strip Symbolic Information (−s Option) . . . . . . . . . . 7-15
7.4.16 Define Stack Size (−stack size Option) . . . . . . . . . . 7-15
7.4.17 Introduce an Unresolved Symbol (−u symbol Option) . . . . . . . . . . 7-16
7.4.18 Display a Message When an Undefined Output Section Is Created (−w Option) . . . . . . . . . . 7-16
7.4.19 Exhaustively Read and Search Libraries (−x and −priority Options) . . . . . . . . . . 7-17
7.4.20 Generate XML Link Information File (−−xml_link_info Option) . . . . . . . . . . 7-18
7.5 Linker Command Files . . . . . . . . . . 7-19
7.5.1 Reserved Names in Linker Command Files . . . . . . . . . . 7-21
7.5.2 Constants in Linker Command Files . . . . . . . . . . 7-21
7.6 Object Libraries . . . . . . . . . . 7-22
7.7 The MEMORY Directive . . . . . . . . . . 7-24
7.7.1 Default Memory Model . . . . . . . . . . 7-24
7.7.2 MEMORY Directive Syntax . . . . . . . . . . 7-24
7.8 The SECTIONS Directive . . . . . . . . . . 7-29
7.8.1 SECTIONS Directive Syntax . . . . . . . . . . 7-29
7.8.2 Allocation . . . . . . . . . . 7-32
7.8.3 Specifying Input Sections . . . . . . . . . . 7-36
7.8.4 Allocating an Archive Member to an Output Section . . . . . . . . . . 7-38
7.9 Specifying a Section’s Run-Time Address . . . . . . . . . . 7-39
7.9.1 Specifying Load and Run Addresses . . . . . . . . . . 7-39
7.9.2 Uninitialized Sections . . . . . . . . . . 7-40
7.9.3 Referring to the Load Address by Using the .label Directive . . . . . . . . . . 7-40
7.10 Using UNION and GROUP Statements . . . . . . . . . . 7-43
7.10.1 Overlaying Sections With the UNION Statement . . . . . . . . . . 7-43
7.10.2 Grouping Output Sections Together . . . . . . . . . . 7-45
7.11 Overlaying Pages . . . . . . . . . . 7-46

7.11.1 Using the MEMORY Directive to Define Overlay Pages . . . . . . . . . . . . . . . . . . 7-46
7.11.2 Example of Overlay Pages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-47
7.11.3 Using Overlay Pages With the SECTIONS Directive . . . . . . . . . . . . . . . . . . . . 7-48
7.11.4 Memory Allocation for Overlaid Pages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-49
7.12 Special Section Types (DSECT, COPY, and NOLOAD) . . . . . . . . . . . . . . . . . . . . . . . . 7-50
7.13 Default Allocation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-51
7.13.1 Output Section Formation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-51
7.13.2 Default Allocation Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-52
7.14 Assigning Symbols at Link Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-53
7.14.1 Syntax of Assignment Statements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-53
7.14.2 Assigning the SPC to a Symbol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-53
7.14.3 Assignment Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-54
7.14.4 Symbols Defined by the Linker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-56
7.14.5 Assigning Exact Start, End, and Size Values of a Section to a Symbol . . . . . 7-57
7.14.6 Why the Dot Operator Does Not Always Work . . . . . . . . . . . . . . . . . . . . . . . . . 7-57
7.14.7 Address and Dimension Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-58
7.15 Creating and Filling Holes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-61
7.15.1 Initialized and Uninitialized Sections . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-61
7.15.2 Creating Holes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-61
7.15.3 Filling Holes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-63
7.15.4 Explicit Initialization of Uninitialized Sections . . . . . . . . . . . . . . . . . . . . . . . . . 7-64
7.16 Linker-Generated Copy Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-65
7.16.1 A Current Boot-Loaded Application Development Process . . . . . . . . . . . . . . 7-65
7.16.2 An Alternative Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-66
7.16.3 Overlay Management Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-67
7.16.4 Generating Copy Tables Automatically with the Linker . . . . . . . . . . . . . . . . . . 7-68
7.16.5 The table() Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-69
7.16.6 Boot-Time Copy Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-69
7.16.7 Using the table( ) Operator to Manage Object Components . . . . . . . . . . . . . . 7-70
7.16.8 Copy Table Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-70
7.16.9 General Purpose Copy Routine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-72
7.16.10 Linker Generated Copy Table Sections and Symbols . . . . . . . . . . . . . . . . . . . . 7-76
7.16.11 Splitting Object Components and Overlay Management . . . . . . . . . . . . . . . . 7-77
7.17 Partial (Incremental) Linking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-80
7.18 Linking C Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-82
7.18.1 Run-Time Initialization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-82
7.18.2 Object Libraries and Run-Time Support . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-82
7.18.3 Setting the Size of the Stack and Heap Sections . . . . . . . . . . . . . . . . . . . . . . . 7-83
7.18.4 Autoinitialization of Variables at Run Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-83
7.18.5 Autoinitialization of Variables at Load Time . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-84
7.18.6 The −c and −cr Linker Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-85
7.19 Linker Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-86

**8 Absolute Lister Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-1**
*Explains how to invoke the absolute lister to obtain a listing of the absolute addresses of an object file.*
8.1 Producing an Absolute Listing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-2

8.2 Invoking the Absolute Lister . . . . . . . . . . 8-3
8.3 Absolute Lister Example . . . . . . . . . . 8-5

**9 Cross-Reference Lister Description . . . . . . . . . . 9-1**

*Explains how to invoke the cross-reference lister to obtain a listing of symbols, their definitions, and their references in the linked source files.*

9.1 Producing a Cross-Reference Listing . . . . . . . . . . 9-2
9.2 Invoking the Cross-Reference Lister . . . . . . . . . . 9-3
9.3 Cross-Reference Listing Example . . . . . . . . . . 9-4

**10 Object File Utilities Descriptions . . . . . . . . . . 10-1**

*Explains how to invoke the object file display utility , the name utility, and the strip utility.*

10.1 Invoking the Object File Display Utility . . . . . . . . . . 10-2
10.2 XML Tag Index . . . . . . . . . . 10-3
10.3 Example XML Consumer . . . . . . . . . . 10-9
10.3.1 The Main Application . . . . . . . . . . 10-9
10.3.2 xml.h { Declaration of the XMLEntity Object . . . . . . . . . . 10-12
10.3.3 xml.cpp { Definition of the XMLEntity Object . . . . . . . . . . 10-13
10.4 Invoking the Name Utility . . . . . . . . . . 10-16
10.5 Invoking the Strip Utility . . . . . . . . . . 10-17

**11 Hex-Conversion Utility Description . . . . . . . . . . 11-1**

*Explains how to invoke the hex utility to convert a COFF object file into one of several standard hexadecimal formats suitable for loading into an EPROM programmer.*

11.1 The Hex-Conversion Utility’s Role in the Software Development Flow . . . . . . . . . . 11-2
11.2 Invoking the Hex-Conversion Utility . . . . . . . . . . 11-3
11.2.1 Invoking the Hex-Conversion Utility From the Command Line . . . . . . . . . . 11-3
11.2.2 Invoking the Hex-Conversion Utility With a Command File . . . . . . . . . . 11-6
11.3 Understanding Memory Widths . . . . . . . . . . 11-8
11.3.1 Target Width . . . . . . . . . . 11-9
11.3.2 Specifying the Memory Width . . . . . . . . . . 11-9
11.3.3 Partitioning Data Into Output Files . . . . . . . . . . 11-10
11.3.4 Specifying Word Order for Output Words . . . . . . . . . . 11-13
11.4 The ROMS Directive . . . . . . . . . . 11-15
11.4.1 When to Use the ROMS Directive . . . . . . . . . . 11-17
11.4.2 An Example of the ROMS Directive . . . . . . . . . . 11-18
11.5 The SECTIONS Directive . . . . . . . . . . 11-21
11.6 Excluding a Specified Section . . . . . . . . . . 11-23
11.7 Assigning Output Filenames . . . . . . . . . . 11-24
11.8 Image Mode and the −fill Option . . . . . . . . . . 11-26
11.8.1 Generating a Memory Image . . . . . . . . . . 11-26
11.8.2 Specifying a Fill Value . . . . . . . . . . 11-27
11.8.3 Steps to Follow in Image Mode . . . . . . . . . . 11-27
11.9 Building a Table for an On-Chip Boot Loader . . . . . . . . . . 11-28

11.9.1 Description of the Boot Table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-28
11.9.2 The Boot Table Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-28
11.9.3 Controlling the Boot Loader . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-29
11.9.4 Booting From a Device Peripheral . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-31
11.9.5 Setting the Entry Point for the Boot Table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-31
11.9.6 Using the C28x Boot Loader . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-32
11.10 Controlling the ROM Device Address . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-38
11.11 Object Formats . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-39
11.11.1 ASCII-Hex Object Format (−a Option) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-40
11.11.2 Intel MCS-86 Object Format (−i Option) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-41
11.11.3 Motorola-S Object Format (−m Option) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-42
11.11.4 TI-Tagged SDSMAC Object Format (−t Option) . . . . . . . . . . . . . . . . . . . . . . . 11-43
11.11.5 Extended Tektronix Object Format (−x Option) . . . . . . . . . . . . . . . . . . . . . . . 11-44
11.12 Hex-Conversion Utility Error Messages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-45

**A Common Object File Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-1**

*Contains supplemental technical data about the internal format and structure of COFF object files. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-1*

A.1 COFF File Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-2
A.2 File Header Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-4
A.3 Optional File Header Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-5
A.4 Section Header Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-6
A.5 Structuring Relocation Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-9
A.6 Symbol Table Structure and Content . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-11
A.6.1 Special Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-12
A.6.2 Symbol Name Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-13
A.6.3 String Table Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-13
A.6.4 Storage Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-14
A.6.5 Symbol Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-14
A.6.6 Section Number . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-15
A.6.7 Auxiliary Entries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-15

**B Symbolic Debugging Directives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-1**

*Discusses symbolic debugging directives that the TMS320C28x C/C++ compiler uses.*

B.1 DWARF Debugging Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-2
B.2 COFF Debugging Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-3
B.3 Debug Directive Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-4

**C XML Link Information File Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-1**

*Discusses the xml_link_info file contents including file element types and document elements.*

C.1 XML Information File Element Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-2
C.2 Document Elements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-3

C.2.1 Header Elements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-3
C.2.2 Input File List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-4
C.2.3 Object Component List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-5
C.2.4 Logical Group List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-6
C.2.5 Placement Map . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-9
C.2.6 Symbol Table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-11

**D Glossary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . D-1**

*Defines terms and acronyms used in this book.*

# Figures

1−1. TMS320C28x Software Development Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1-2
2−1. Partitioning Memory Into Logical Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-3
2−2. Object Code Generated by the File in Example 2−1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-10
2−3. Combining Input Sections to Form an ExecuObject Module . . . . . . . . . . . . . . . . . . . . . . . . 2-12
3−1. The Assembler in the TMS320C28x Software Development Flow . . . . . . . . . . . . . . . . . . 3-3
4−1. The .space and .bes Directives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-10
4−2. The .field Directive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-11
4−3. Initialization Directives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-12
4−4. The .align Directive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-14
4−5. Allocating .bss Blocks Within a Page . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-28
4−6. The .field Directive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-44
4−7. The .usect Directive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-82
6−1. The Archiver in the TMS320C28x Software Development Flow . . . . . . . . . . . . . . . . . . . . . 6-3
7−1. The Linker in the TMS320C28x Software Development Flow . . . . . . . . . . . . . . . . . . . . . . 7-3
7−2. Memory Map Defined in Example 7−3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-26
7−3. Section Allocation Defined in Example 7−4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-31
7−4. Run-Time Execution of Example 7−6 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-42
7−5. Memory Allocation Shown in Example 7−7 and Example 7−8 . . . . . . . . . . . . . . . . . . . . . 7-44
7−6. Overlay Pages Defined in Example 7−10 and Example 7−11 . . . . . . . . . . . . . . . . . . . . . . 7-49
7−7. Autoinitialization at Run Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-83
7−8. Autoinitialization at Load Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-84
8−1. Absolute Lister Development Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-2
8−2. module1.lst . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-9
8−3. module2.lst . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-9
9−1. Cross-Reference Lister Development Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-2
11−1. The Hex-Conversion Utility in the TMS320C28x Software Development Flow . . . . . . . . 11-2
11−2. Hex-Conversion Utility Process Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-8
11−3. COFF Data and Memory Widths . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-10
11−4. Data, Memory, and ROM Widths . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-12
11−5. Varying the Word Order . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-14
11−6. The infile.out File Partitioned Into Four Output Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-18
11−7. ASCII-Hex Object Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-40
11−8. Intel MCS86 Hexadecimal Object Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-41
11−9. Motorola-S Object Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-42
11−10. TI-Tagged Object Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-43
11−11. Extended Tektronix Object Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-44

A−1. COFF File Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-2
A−2. Sample COFF Object File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-3
A−3. Section Header Pointers for the .text Section . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-8
A−4. Symbol-Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-11
A−5. String-Entries for Sample Symbol Names . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-13

# Tables

3−1. Order of Precedence of Operators Used in Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . 3-26
3−2. Built-In Mathematical Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-29
3−3. Symbol Attributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-36
3−4. Smart Encoding for Efficiency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-37
3−5. Smart Encoding Intuitively . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-38
3−6. Instructions That Avoid Smart Encoding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-38
3−7. Non-TMS320C27x Instructions Supported in the C27x Object Mode . . . . . . . . . . . . . . . . 3-42
4−1. Assembler Directives Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-2
5−1. Functions and Return Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-8
5−2. Creating Macros . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-23
5−3. Manipulating Substitution Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-23
5−4. Conditional Assembly . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-23
5−5. Producing Assembly-Time Messages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-24
5−6. Formatting the Listing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-24
7−1. Linker Options Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-5
7−2. Groups of Operators Used in Expressions on Order of Precedence . . . . . . . . . . . . . . . . . 7-55
9−1. Symbol Attributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-5
10−1. XML Tag Index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-3
11−1. Hex Conversion Utility Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-4
11−2. Boot-Loader Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-29
11−3. Boot Source Formats . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-32
11−4. Boot Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-32
11−5. Options for Specifying Hex-Conversion Formats . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-39
A−1. File Header Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-4
A−2. File Header Flags (Bytes 18 and 19) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-4
A−3. Optional File Header Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-5
A−4. Section Header Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-6
A−5. Section Header Flags (Bytes 40 Through 43) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-7
A−6. Relocation Entry Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-9
A−7. Relocation Types (Bytes 8 and 9) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-10
A−8. Symbol-Entry Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-12
A−9. Special Symbols in the Symbol Table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-12
A−10. Symbol Storage Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-14
A−11. Section Numbers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-15
A−12. Section Format for Auxiliary Entries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-15
B−1. Symbolic Debugging Directives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-4

# Examples

2−1. Using Sections Directives . . . . . 2-9
2−2. An Example of Code That Generates Relocation Entries . . . . . 2-14
2−3. Relocation Expression . . . . . 2-15
3−1. Local Labels of the Form $n . . . . . 3-19
3−2. Local Labels . . . . . 3-20
3−3. Using Symbolic Constants Defined on Command Line . . . . . 3-22
3−4. Portion of an Assembler Listing . . . . . 3-33
3−5. Viewing Assembly Variables as C Types . . . . . 3-34
3−6. An Assembler Cross-Reference Listing . . . . . 3-36
4−1. Sections Directives . . . . . 4-9
5−1. Macro Definition, Call, and Expansion . . . . . 5-4
5−2. Calling a Macro With Varying Numbers of Arguments . . . . . 5-6
5−3. The .asg Directive . . . . . 5-6
5−4. The .eval Directive . . . . . 5-7
5−5. Using Built-In Substitution Symbol Functions . . . . . 5-9
5−6. Recursive Substitution . . . . . 5-9
5−7. Using the Forced Substitution Operator . . . . . 5-10
5−8. Using Subscripted Substitution Symbols to Redefine an Instruction . . . . . 5-11
5−9. Using Subscripted Substitution Symbols to Find Substrings . . . . . 5-12
5−10. The .loop/.break/.endloop Directives . . . . . 5-15
5−11. Nested Conditional Assembly Directives . . . . . 5-15
5−12. Built-In Substitution Symbol Functions in a Conditional Assembly Block . . . . . 5-15
5−13. Unique Labels in a Macro . . . . . 5-16
5−14. Producing Messages in a Macro . . . . . 5-18
5−15. Using Nested Macros . . . . . 5-21
5−16. Using Recursive Macros . . . . . 5-22
7−1. Linker Command File . . . . . 7-19
7−2. Command File With Linker Directives . . . . . 7-20
7−3. The MEMORY Directive . . . . . 7-25
7−4. The SECTIONS Directive . . . . . 7-31
7−5. The Most Common Method of Specifying Section Contents . . . . . 7-36
7−6. Copying a Section From ROM to RAM . . . . . 7-41
7−7. The UNION Statement . . . . . 7-43
7−8. Separate Load Addresses for UNION Sections . . . . . 7-43
7−9. Allocate Sections Together . . . . . 7-45
7−10. Memory Directive With Overlay Pages . . . . . 7-47

7−11. SECTIONS Directive Definition for Overlays in 7−10 . . . . . . . . . . 7-48
7−12. Default Allocation for TMS320C28x Devices . . . . . . . . . . 7-51
7−13. Using a UNION for Memory Overlay . . . . . . . . . . 7-67
7−14. Produce Address for Linker Generated Copy Table . . . . . . . . . . 7-68
7−15. Linker Command File to Manage Object Components . . . . . . . . . . 7-70
7−16. TMS320C28x cpy_tbl.h File . . . . . . . . . . 7-71
7−17. Run-Time-Support cpy_tbl.c File . . . . . . . . . . 7-73
7−18. The cpy_utils.asm File . . . . . . . . . . 7-74
7−19. Controlling the Placement of the Linker-Generated Copy Sections . . . . . . . . . . 7-76
7−20. Creating a Copy to Access a Split Object Component . . . . . . . . . . 7-78
7−21. Split Object Component Driver . . . . . . . . . . 7-78
7−22. Linker Command File, demo.cmd . . . . . . . . . . 7-87
7−23. Output Map File, demo.map . . . . . . . . . . 7-88
11−1. A ROMS Directive Example . . . . . . . . . . 11-18
11−2. Map File Output From 11−1 Showing Memory Ranges . . . . . . . . . . 11-19
11−3. Sample Command File for Booting From 8-Bit SPI Boot . . . . . . . . . . 11-33
11−4. Sample Hex Convertor Out File for Booting From 8-Bit SPI Boot . . . . . . . . . . 11-34
11−5. Sample Command File for C28x 16-Bit Parallel Boot GP I/O . . . . . . . . . . 11-35
11−6. Sample Hex Convertor Out File for C28x 16-Bit Parallel Boot GP I/O . . . . . . . . . . 11-35
11−7. Sample Command File for Booting From 8-Bit SCI Boot . . . . . . . . . . 11-36
11−8. Sample Hex Convertor Out File for Booting From 8-Bit SCI Boot . . . . . . . . . . 11-37
C−1. Header Element for the hi.out Output File . . . . . . . . . . C-3
C−2. Input File List for the hi.out Output File . . . . . . . . . . C-4
C−3. Object Component List for the fl−4 Input File . . . . . . . . . . C-5
C−4. Logical Group List for the fl−4 Input File . . . . . . . . . . C-8
C−5. Placement Map for the fl−4 Input File . . . . . . . . . . C-10
C−6. Symbol for the fl−4 Input File . . . . . . . . . . C-11

# Notes

Default Section Directive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-4
Expression Cannot Be Larger Than Space Reserved . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-15
Octal Numbers Not Accepted With C2xlp Syntax Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-14
Differences in Precedence From Other TMS320 Assemblers . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-25
Labels and Comments in Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-2
Directives That Initialize Constants When Used in a.struct/.endstruct Sequence . . . . . . . . . . . . 4-11
Ending a Macro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-40
Creating a Listing File (−l Option) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-54
The .struct Directive Does Not Allocate Memory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-74
Directives That Can Appear in a .struct/.endstruct Sequence . . . . . . . . . . . . . . . . . . . . . . . . . . 4-75
Naming Library Members . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-5
The −a and −r Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-7
Filling Memory Ranges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-27
Binding Is Incompatible With Alignment and Named Memory . . . . . . . . . . . . . . . . . . . . . . . . . . 7-33
You Cannot Specify Addresses for Sections Within a GROUP . . . . . . . . . . . . . . . . . . . . . . . . . . 7-45
Overlay Section and Overlay Page Are Not the Same . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-46
Linker Command File Operator Equivalencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-58
Filling Sections . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-64
Linking the .stack Section . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-83
Boot Loader . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-85
The TI-Tagged Format Is 16 Bits Wide . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-11
When the −order Option Applies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-13
Sections Generated by the C/C++ Compiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-21
Defining the Ranges of Target Memory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-26
Valid Entry Points . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-31

Chapter 1

# Introduction to the Software Development Tools

The TMS320C28x™ DSP is supported by a set of software development tools that includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities.

This overview shows how these tools fit into the general software tools development flow and describes each tool briefly. The TMS320C28x assembly language development tools are:

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

**Topic** **Page**

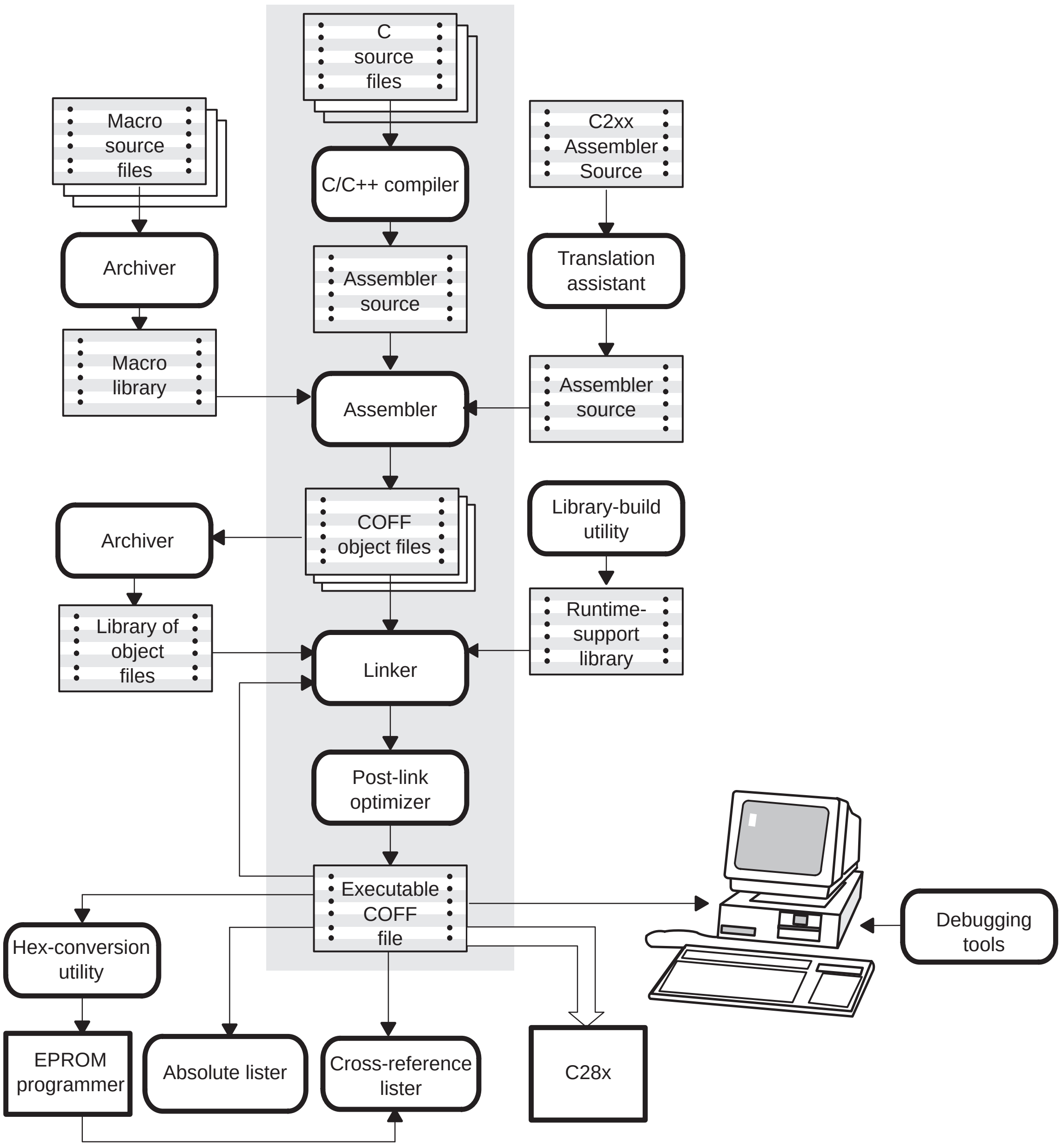
**1.1 Software Development Tools Overview . . . . . . . . . . . . . . . . . . . . . . . . . 1-2**

**1.2 Tools Descriptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1-3**

## 1.1 Software Development Tools Overview

Figure 1−1 shows the TMS320C28x software development flow. The shaded portion highlights the most common development path; the other portions are optional. They are peripheral functions that enhance development.

*Figure 1−1. TMS320C28x Software Development Flow*



## 1.2 Tools Descriptions

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

- The **C/C++ compiler** accepts C and C++ source code and produces TMS320C28x assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:

  - The shell program enables you to compile, assemble, and link source modules in one step.
  - The optimizer modifies code to improve the efficiency of C 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 *TMS320C28x 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 *TMS320C28x 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 *TMS320C28x 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 **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 11, *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 **TMS320C28x** device.
- ❑ You can use one of several debugging tools to refine and correct your code. Available products include:
  - ■ An instruction-accurate and clock-accurate software simulator
  - ■ An XDS emulator
  - ■ An evaluation module (EVM)

  For information about these debugging tools, see the *TMS320C28x 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 TMS320C28x™ 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 . . . . . 2-2**

**2.2 How the Assembler Handles Sections . . . . . 2-4**

**2.3 How the Linker Handles Sections . . . . . 2-11**

**2.4 Relocation . . . . . 2-14**

**2.5 Loading a Program . . . . . 2-17**

**2.6 Symbols in a COFF File . . . . . 2-18**

## 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:

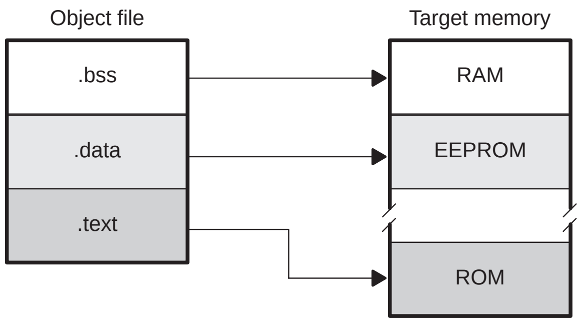
| | |
|---|---|
| **Initialized sections** | contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized. |
| **Uninitialized sections** | reserve space in the memory map for uninitialized data. The .bss section is 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 TMS320C28x memory; they are usually allocated into RAM. These sections have no actual contents in the object file; they simply reserve memory. A program can use this space at run time for creating and storing variables.

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

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

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

The syntaxes for these directives are:

| | |
|---|---|
| | **.bss** *symbol*, *size in words* [, *blocking flag*] [, *alignment flag*] |
| *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.<br>❑ 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.<br>❑ 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 TMS320C28x memory when the program is loaded. Each initialized section is independently relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these 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 7-29 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 7-29 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. |
| **Field 2** | contains the section program counter. |
| **Field 3** | contains the object code. |
| **Field 4** | contains the original source statement. |

See section 3.11, *Source Listings*, on page 3-31 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 00000000                   .data
  5 00000000 0011  coeff      .word   011h, 022h, 033h
    00000001 0022
    00000002 0033
  6
  7                **************************************************
  8                ** Reserve space in .bss for a variable.       **
  9                **************************************************
 10 00000000                   .bss    buffer, 10
 11
 12                **************************************************
 13                **              Still in .data                 **
 14                **************************************************
 15 00000003 0123  ptr        .word   0123h
 16
 17                **************************************************
 18                **    Assemble code into the .text section     **
 19                **************************************************
 20 00000000                   .text
 21 00000000 28A1  add:       mov     ar1, #0Fh
    00000001 000F
 22 00000002 0BA1  aloop:     dec     ar1
 23 00000003 0009             banz    aloop, ar1--
    00000004 FFFF
 24
 25                ****************************************************
 26                ** Another initialized table into .data          **
 27                ****************************************************
 28 00000004                   .data
 29 00000004 00AA  ivals      .word   0AAh, 0BBh, 0CCh
    00000005 00BB
    00000006 00CC
 30
 31                **************************************************
 32                ** Define another section for more variables.   **
 33                **************************************************
 34 00000000       var2       .usect  ”newvars”, 1
 35 00000001       inbuf      .usect  ”newvars”, 7
 36
 37                **************************************************
 38                ** Assemble more code into .text.              **
 39                **************************************************
 40 00000005                   .text
 41 00000005 28A1  end_mpy:   mov     ar1, #0Ah
    00000006 000A
 42 00000007 33A1  mloop:     mpy     p,t,ar1
 43 00000008 28AC             mov     t, #0Ah
    00000009 000A
 44 0000000a 3FA1             mov     ar1, p
 45 0000000b 6BFA             sb      end_mpy, OV
```

Field 1  Field 2  Field 3  Field 4

As Figure 2−2 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 | 28A1 | .text |
| 21 | 000F | |
| 22 | 0BA1 | |
| 23 | 0009 | |
| 23 | FFFF | |
| 41 | 28A1 | |
| 41 | 000A | |
| 42 | 33A1 | |
| 43 | 28AC | |
| 43 | 000A | |
| 44 | 3FA1 | |
| 45 | 6BFB | |
| 10 | No data | .bss |
| | 10 words preserved | |
| 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.

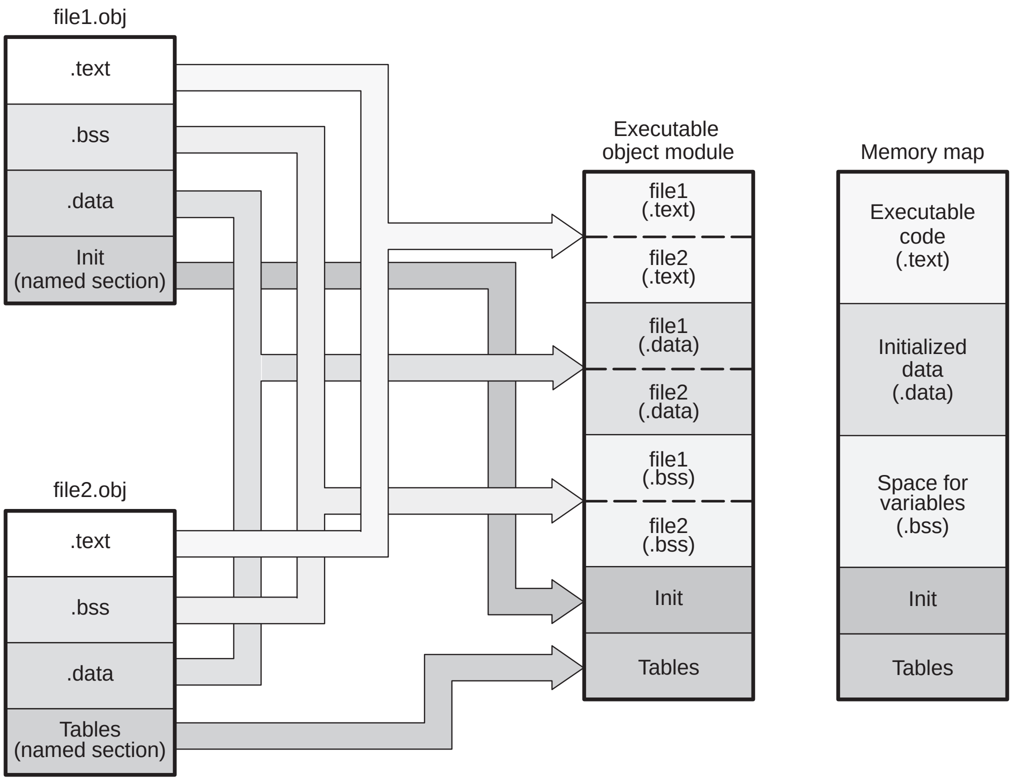
It is not always necessary to use linker directives. If you do not use them, the linker uses the target processor's default allocation algorithm described in section 7.13, *Default Allocation*, on page 7-51. When you *do* use linker directives, you must specify them in a linker command file.

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 7-83). For more information on the .const and .cinit sections, see the *TMS320C28x 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 7-24 and section 7.8, *The SECTIONS Directive*, on page 7-29.

## 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 TMS320C28x code segment that generates relocation entries.

*Example 2−2. An Example of Code That Generates Relocation Entries*

```
                        .global X
00000000                .text
00000000 0080'          LC      Y               ; Generates a Relocation Entry
00000001 0004
00000002 28A1!          MOV     AR1,#X          ; Generates a Relocation Entry
00000003 0000
00000004 7621  Y:       IDLE
```

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

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

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

Sometimes an expression contains more than one relocatable symbol, or cannot be evaluated at assembly time. In this case, the assembler encodes the entire expression in the object file. After determining the addresses of the symbols, the linker computes the value of the expression as shown in Example 2−3.

*Example 2−3. Relocation Expression*

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

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

```
00000000 FF20           mov     ACC, #(sym2−sym1)
00000001 0100
```

**Note: Expression Cannot Be Larger Than Space Reserved**

If the value of an expression is larger, in bits, than the space reserved for it, the linker produces an error message.

Each section in a COFF object file has a table of relocation entries. The table contains one relocation entry for each relocatable reference in the section. The linker usually removes relocation entries after it uses them. This prevents the output file from being relocated again (if it is relinked or when it is loaded). A file that contains no relocation entries is an absolute file (all its addresses are absolute addresses). If you want the linker to retain relocation entries, invoke the linker with the −r option.

### 2.4.1 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 7-41.

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 7-39.

## 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 TMS320C28x 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.

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

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

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

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 .global definition of z says that it is an undefined symbol in this module and is defined in another module. The .global definition of a says that it is an external symbol defined in this module and that other modules can reference a.

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

The linker 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 TMS320C28x™ assembler translates assembly language source files into machine language object files. These files are in common object file format (COFF), which is discussed in Chapter , *Introduction to Common Object File Format*, and Appendix , *Common Object File Format*. Source files can contain the following assembly language elements:

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

**Topic** **Page**

**3.1 Assembler Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-2**

**3.2 The Assembler’s Role in the Software Development Flow . . . . . . . . 3-3**

**3.3 Invoking the Assembler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-4**

**3.4 Naming Alternate Directories for Assembler Input . . . . . . . . . . . . . . . 3-7**

**3.5 Source Statement Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-10**

**3.6 Constants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-13**

**3.7 Character Strings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-17**

**3.8 Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-18**

**3.9 Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-25**

**3.10 Built-In Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-29**

**3.11 Source Listings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-31**

**3.12 Debugging Assembly Source . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-34**

**3.13 Cross-Reference Listing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-36**

**3.14 Smart Encoding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-37**

**3.15 C-Type Symbolic Debugging for Assembly Variables (−mg option) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-39**

**3.16 TMS320C28x Assembler Modes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-41**

## 3.1 Assembler Overview

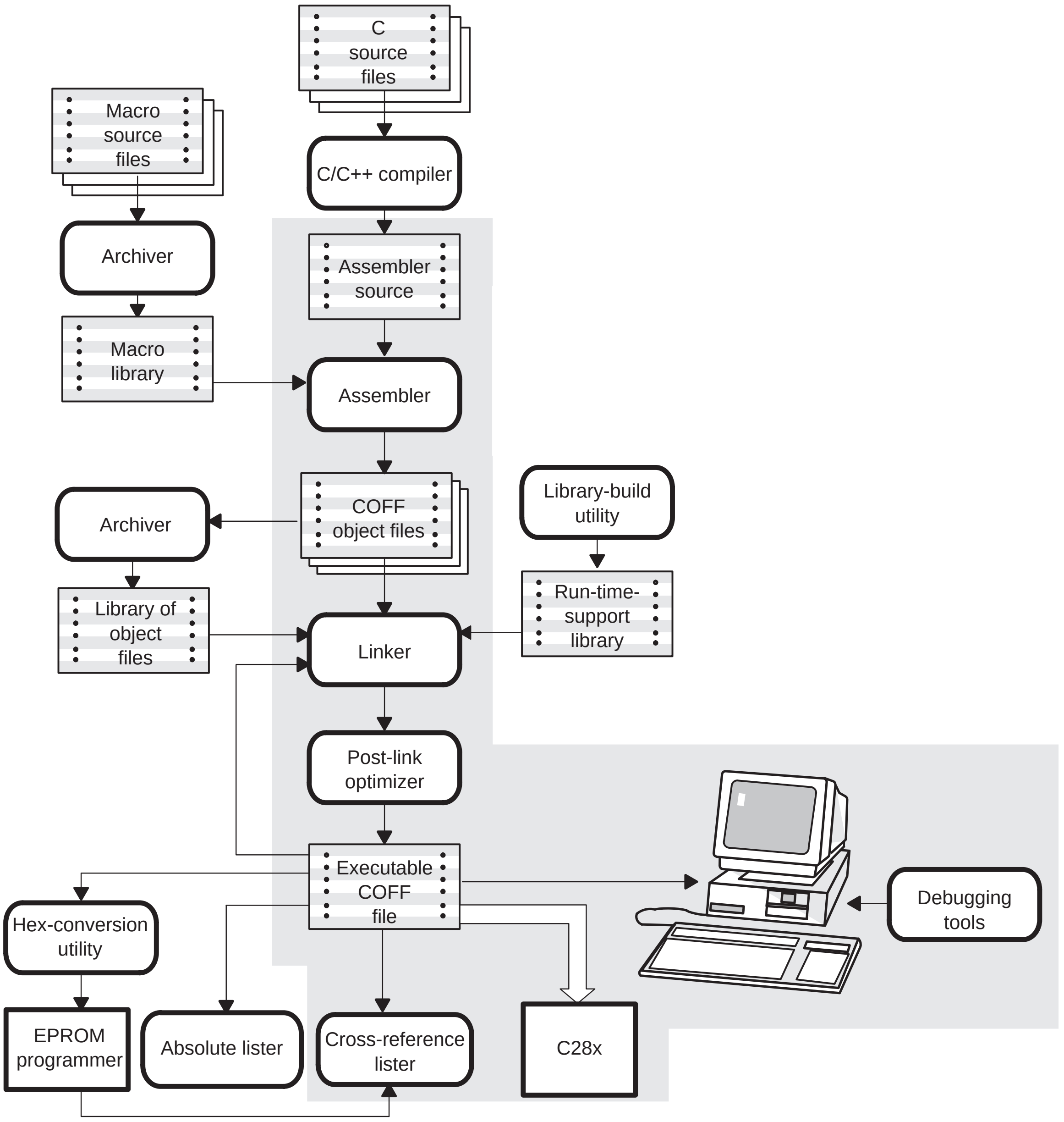
The 2-pass assembler does the following tasks:

- 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

## 3.2 The Assembler's Role in the Software Development Flow

Figure 3−1 illustrates the assembler's role in the software development flow. The shaded portion highlights the most common assembler development path. The assembler accepts assembly language source files.

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



## 3.3 Invoking the Assembler

Invoke the assembler through the compiler, enter the following:

**cl2000** *version* [*input file* [*object file* [*listing file*] ] ] [*options*]

| | |
|---|---|
| **cl2000** | is the command that invokes the assembler through the compiler. The compiler considers any file with an .asm extension to be an assembly file and calls the assembler. |
| *version* | refers to the target processor for which the source file is assembled. The valid versions are −v28 for the TMS320C28x processor and −v27 for the TMS320C27x processor. The version is required; the assembler issues an error if version is not specified.<br><br>If both −v27 and −v28 are specified, the assembler ignores the second version and issues a warning. For more information, see section 3.16, *TMS320C28x Assembler Modes*, on page 3-41. |
| *input file* | names the assembly language source file. |
| *options* | identify the assembler options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen. 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. |

The valid assembler options are as follows:

| | |
|---|---|
| **−aa** | 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. |
| **−ac** | 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). |
| **−ad** | **−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 3.8.4, *Defining Symbolic Constants (−d Option)*, on page 3-21. |
| **−ahc** | **−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. |

| | |
|---|---|
| **−ahi** | **−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. |
| **−al** | (lowercase L) produces a listing file with the same name as the input file with a *.lst* extension. |
| **−as** | puts all defined symbols in the object file’s symbol table. The assembler usually puts only global symbols into the symbol table. When you use −s, symbols defined as labels or as assembly-time constants are also placed in the table. |
| **−au** | **−u***name* undefines the predefined constant *name*, which overrides any −d options for the specified constant. |
| **−ax** | produces a cross-reference table and appends it to the end of the listing file; it also adds cross-reference information to the object file for use by the cross-reference utility. If you do not request a listing file but use the −x option, the assembler creates a listing file automatically, naming it with the same name as the input file with a *.lst* extension. |
| **−g** | enables assembler source debugging in the C source debugger. Line information is output to the COFF file for every line of source in the assembly language source file. You cannot use the −g option on assembly code that contains .line directives. See section 3.12, *Debugging Assembly Source*, on page 3-34 for more information. |
| **−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 3.4.1, *Using the −i Assembler Option*, on page 3-7. |
| **−m20** | accepts C2xLP assembly instructions and encodes them as equivalent C28x instructions. The −m20 option implies the −v28 option. For more information, see section 3.16, *TMS320C28x Assembler Modes*, on page 3-41. |
| **−m27** | accepts C27x instruction syntax. The −m27 option implies the −v28 option. For more information, see section 3.16, *TMS320C28x Assembler Modes*, on page 3-41. |

| | |
|---|---|
| **−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). |

## 3.4 Naming Alternate Directories for Assembler Input

The .copy, .include, and .mlib directives tell the assembler to use code from external files. The .copy and .include directives tell the assembler to read source statements from another file, and the .mlib directive names a library that contains macro functions. Chapter 4, *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. If *filename* begins with a number the double quotes are required. The filename may be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in the following locations in the order given:

1) The directory that contains the current source file. The current source file is the file being assembled when the .copy, .include, or .mlib directive is encountered.
2) Any directories named with the `−I` assembler option
3) Any directories named with the C2000_A_DIR or 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 3.4.1) or the C2000_A_DIR and A_DIR environment variables (described in section 3.4.2).

### 3.4.1 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:

> **cl2000 −v28** `−I`*pathname* *source filename* [*other options*]

Each `−I` option names one pathname. There is no limit to the number of paths you can specify. 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; source.asm contains the following directive statement:

```
.copy ”copy.asm”
```

Assume the following paths for the copy.asm file:

| | |
|---|---|
| Windows: | c:\320tools\files\copy.asm |
| UNIX: | /320tools/files/copy.asm |

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

| Operating System | Enter |
|---|---|
| Windows | **`cl2000 -v28 −Ic:\320tools\files source.asm`** |
| UNIX | **`cl2000 -v28 −I/320tools/files source.asm`** |

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

### 3.4.2 Using the C2000_A_DIR and A_DIR Environment Variables

An environment variable is a system symbol that you define and assign a string to. The assembler uses the C2000_A_DIR and A_DIR environment variables to name alternate directories that contain .copy/.include files or macro libraries. The command syntax for assigning the environment variable is as follows:

The assembler looks for the C2000_A_DIR environment variable first and then reads and processes it. If it does not find this variable, it reads the A_DIR environment variable and processes it. If both variables are set, the settings of the processor-specific variable are used. The processor-specific variable is useful when you are using Texas Instruments tools for different processors at the same time.

If the assembler does not find C2000_A_DIR and A_DIR, it then searches for C2000_C_DIR and C_DIR. See the *TMS320C28x Optimizing Compiler User’s Guide* for details on C2000_C_DIR and C_DIR.

| Operating System | Enter |
|---|---|
| Windows | **set A_DIR=** *pathname*$_1$*;pathname*$_2$*; . . .* |
| UNIX (Bourne shell) | **A_DIR=”***pathname*$_1$**;***pathname*$_2$**;** *. . .***”;**<br>**export A_DIR** |

The *pathnames* are directories that contain copy/include files or macro libraries. You can separate the pathnames with a semicolon or with blanks. In assembly source, you can use the .copy, .include, or .mlib directive without specifying path information. If the assembler does not find the file in the directory that contains the current source file or in directories named by the −I option, it searches the paths named by the environment variable.

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

```
.copy ”copy1.asm”
.copy ”copy2.asm”
```

Assume the following paths for the files:

| | |
|---|---|
| Windows: | c:\tools\files\copy1.asm<br>c:\dsys\copy2.asm |
| UNIX: | /tools/files/copy1.asm<br>/dsys/copy2.asm |

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

| Operating System | Enter |
|---|---|
| Windows | `set A_DIR=c:\dsys`<br>`cl2000 −v28 −Ic:\tools\files source.asm` |
| UNIX (Bourne Shell) | `A_DIR=”/dsys”; export A_DIR`<br>`cl2000 −v28 −I=/tools/files source.asm` |

The assembler first searches for copy1.asm and copy2.asm in the current directory because source.asm is in the current directory. Then the assembler searches in the directory named with the −I option and finds copy1.asm. Finally, the assembler searches the directory named with A_DIR and finds copy2.asm.

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

| Operating System | Enter |
|---|---|
| Windows | `set A_DIR=` |
| UNIX | `unset A_DIR` |

## 3.5 Source Statement Format

TMS320C28x assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. A source statement can contain four ordered fields (label, mnemonic, operand list, and comment).

The general syntax for source statements is as follows:

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

Following are examples of source statements:

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

The C28x assembler reads up to 200 characters per line. Any characters beyond 200 are truncated. Keep the operational part of your source statements (that is, everything other than comments) less than 200 characters in length for correct assembly. Your comments can extend beyond the 200 character limit, but the truncated portion is not included in the listing file.

Follow these guidelines:

❑ All statements must begin with a label, a blank, an asterisk, or a semicolon.

❑ Labels are optional; if used, they must begin in column 1.

❑ One or more blanks must separate each field. Tab 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.

❑ A mnemonic cannot begin in column one or it will be interpreted as a label.

The following sections describe each of the fields.

### 3.5.1 Label Field

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

When you use a label, its value is the current value of the section program counter (SPC). 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  .set   $   ; $ provides the current value of the SPC
```

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

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

### 3.5.2 Mnemonic Field

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

```
        RPT
||      Inst2 ◄──────  This instruction is repeated.
```

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 .set)
- ❑ A macro directive (such as .macro, .var, or .mexit)
- ❑ A macro call

### 3.5.3 Operand Field

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

- ❑ Symbols (see section 3.8 on page 3-18)
- ❑ Constants (see section 3.6 on page 3-13)
- ❑ Expressions (a combination of constants and symbols; see section 3.9 on page 3-25)

You must separate operands with commas.

### 3.5.4 Comment Field

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

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

## 3.6 Constants

The assembler supports seven types of constants:

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

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

### 3.6.1 Binary Integers

A binary integer constant is a string of up to 32 binary digits (0s and 1s) followed by the suffix B (or b). If fewer than 32 digits are specified, the assembler right-justifies the value and 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 $0F8_{16}$ |

### 3.6.2 Octal Integers

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

| | |
|---|---|
| **10Q** | 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 $96_{16}$ |

**Note: Octal Numbers Not Accepted With C2xlp Syntax Mode**

When the −v28 −m20 options are specified, asm2000 accepts C2xlp source code. The C2xlp assembler interpreted numbers with leading zeros as decimal integers, that is, 010 was interpreted as $10_{10}$. Because of this, when asm2000 is invoked with −v28 −m20 −mw, the assembler issues a warning when it encounters an octal number.

### 3.6.3 Decimal Integers

A decimal integer constant is a string of decimal digits ranging from −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 $19_{16}$ |

### 3.6.4 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 $37AC_{16}$ |

### 3.6.5 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 3.7 on page 3-17 discusses character strings.) A character constant represents a single integer value; a string is a sequence of characters.

### 3.6.6 Assembly-Time Constants

If you use the .set directive (see page 4-70) 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
```

### 3.6.7 Floating-Point Constants

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

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

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

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

## 3.7 Character Strings

A character string is a string of characters enclosed in *double* 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

## 3.8 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 200 alphanumeric characters (A−Z, a−z, 0−9, $, and _). The first character in a symbol cannot be a number, and symbols cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler recognizes ABC, Abc, and abc as three unique symbols. You can override case sensitivity with the −c assembler option (see section 3.3, on page 3-4). 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.

### 3.8.1 Labels

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

Labels can also be used as the operands of .global, .ref, .def, or .bss directives, as shown in the following example:

```
        .global label1

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

### 3.8.2 Local Labels

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

- $n, where n is a decimal digit in the range 0−9. For example, $4 and $1 are valid local labels. See the Local Labels of the Form $n example.
- 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. See the Local Labels of the Form name? example.

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

*Example 3−1. Local Labels of the Form $n*

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

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

       .newblock           ; undefine $1 to use it again.

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

The following code uses local labels illegally:

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

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

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

Local labels are especially useful in macros. If a macro contains a normal label and is called more than once, the assembler issues a multiple-definition error. 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.

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

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 3−2 shows code that declares and used local labels legally

*Example 3−2. Local Labels*

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

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

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

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

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

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

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

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

### 3.8.3 Symbolic Constants

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

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

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

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

The assembler also has several predefined symbolic constants; these are discussed in section 3.8.5, *Predefined Symbolic Constants*, on page 3-22.

### 3.8.4 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:

**cl2000 −v28 −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:

```
cl2000 -v28 -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 3−3 shows how the value.asm file uses these symbols without defining them explicitly.

*Example 3−3. Using Symbolic Constants Defined on Command Line*

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

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

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

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

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.

### 3.8.5 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.

❑ **Processor symbols**, including the following:

| Symbol Name | Description |
|---|---|
| .TMS320C2700 | Set to 1 for C27x (−v27 switch), otherwise 0 |
| .TMS320C2800 | Set to 1 for C28x (−v28 switch), otherwise 0 |
| __LARGE_MODEL | Set to 1 for Large Model (−mf switch), otherwise 0 |

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

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

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

### 3.8.6 Substitution Symbols

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

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

```
    .asg    ”AR1”,  myReg         ;register AR1
    .asg    ”*+XAR2 [2]”,  ARG1   ;first arg
    .asg    ”*+XAR2 [1]”,  ARG2   ;second arg
```

When you are using macros, substitution symbols are important because macro parameters are actually substitution symbols that are assigned a macro argument. The following code shows how substitution symbols are used in macros:

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

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

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

## 3.9 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

### 3.9.1 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 assemblers use a different order of precedence than the TMS320C28x assembler uses. For this reason, different results may be produced from the same source code. The TMS320C28x uses the same order of precedence that the C language uses.
- When asm2000 is invoked with −v28 −m20, the assembler accepts C2xlp source code. A programmer writing code for the C2xlp assembler would assume different precedence than that used by the C28x assembler. Therefore when invoked with the −v28 −m20 −mw options, the C28x assembler issues a warning when it encounters an expression such as a + b << c.

*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.

## 3.9.2 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.

### 3.9.3 Well-Defined Expressions

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

This is an example of a well-defined expression:

```
0x1000+X
```

where X was previously defined as an absolute symbol.

### 3.9.4 Conditional Expressions

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

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

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

### 3.9.5 Legal Expressions

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

When an expression contains more than one relocatable symbol or cannot be evaluated at assembly time, the assembler encodes a relocation expression in the object file that is later evaluated by the linker. If the final value of the expression is larger in bits than the space reserved for it, you will receive an error message from the linker. For more information on relocation expressions, see section 2.4 on page 2-14.

When using the register relative addressing mode, the expression in brackets or parenthesis must be a well-defined expression, as described in section 3.9.3. For example:

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

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

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

- **Example 1**

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

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

- **Example 2**

  All of the following statements are valid.

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

- **Example 3**

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

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

- **Example 4**

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

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

## 3.10 Built-In Functions

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

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

*Table 3−2. Built-In Mathematical Functions*

| Function | Description |
|---|---|
| **$acos**(*x*) | Returns $\cos^{-1}(x)$in range $[0, \pi]$, $x \in [-1, 1]$ |
| **$asin**(*x*) | Returns $\sin^{-1}(x)$in range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$ |
| **$atan**(*x*) | Returns $\tan^{-1}(x)$in range $[-\pi/2, \pi/2]$ |
| **$atan2**(*x,y*) | Returns $\tan^{-1}$(y/x)in range $[-\pi, \pi]$ |
| **$ceil**(*x*) | Returns the smallest integer not less than *x,* as a float. |
| **$cos**(*x*) | Returns the cosine of *x* |
| **$cosh**(*x*) | Returns the hyperbolic cosine of *x* |
| **$cvf**(*n*) | Converts an integer to a float |
| **$cvi**(*x*) | Converts a float to an integer. Returns an integer. |
| **$exp**(*x*) | Returns the exponential function $e^x$ |
| **$fabs**(*x*) | Returns the absolute value $\|x\|$ |
| **$floor**(*x*) | Returns the largest integer not greater than *x*, as a float. |
| **$fmod**(*x, y*) | Returns the floating-point remainder of *x*/*y*, with the same sign as *x* |
| **$int(**x) | Returns 1 if *x* has an integer value; else returns 0. Returns an integer. |
| **$ldexp**(*x,n*) | Multiplies *x* by an integer power of 2. That is, $x \times 2^n$ |
| **$log**(*x*) | Returns the natural logarithm $\ln(x)$, where $x>0$ |
| **$log10**(*x*) | Returns the base-10 logarithm $\log_{10}(x)$, $x>0$ |
| **$max**(*x, y, … z*) | Returns the greatest value from the argument list |
| **$min**(*x, y, … z*) | Returns the smallest value from the argument list |

*Table 3−2. Built-In Mathematical Functions (Continued)*

| **Function** | **Description** |
|---|---|
| **$pow***(x,y)* | Returns $x^y$ |
| **$round***(x)* | Returns *x* rounded to the nearest integer |
| **$sgn***(x)* | Returns the sign of *x*. Returns 1 if *x* is positive, 0 if *x* is zero, and –1 if *x* is negative. Returns an integer. |
| **$sin***(x)* | Returns the sine of *x* |
| **$sinh***(x)* | Returns the hyperbolic sine of *x* |
| **$sqrt***(x)* | Returns the square root of *x*, *x*>=0 |
| **$tan***(x)* | Returns the tangent of *x* |
| **$tanh***(x)* | Returns the hyperbolic tangent of *x* |
| **$trunc(**x) | Returns *x* truncated toward 0 |

The built-in substitution symbol functions are discussed in the section 5.3.2 on page 5-8.

## 3.11 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 3-5).

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 3−4 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 3−4. 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
```

## 3.12 Debugging Assembly Source

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

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

The .asmfunc and .endasmfunc directives (see page 4-25) allow you to name certain areas of your code, and make these areas appear in the debugger as C functions. Contiguous sections of assembly code that are not enclosed by the .asmfunc and .endasmfunc directives are automatically placed in assembler-defined functions named with this syntax:

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

If you want to view your variables as a user-defined type in C code, the types must be declared and the variables must be defined in a C file. This C file can then be referenced in assembly code using the .ref directive (see page 4-46).

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

*Example 3−5. Viewing Assembly Variables as C Types*

*(a) C Program cvar.c*

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

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

*Example 3−5.Viewing Assembly Variables as C Types (Continued)*

*(b) Assembly Program addfive.asm*

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

;-------------------------------------------------------------------------------
; addfive() - Add five to the second data member of _svar
;-------------------------------------------------------------------------------
          .text
          .global addfive
addfive:  .asmfunc
           MOVZ      DP,#_svar+1   ; load the DP with svar's memory page
           ADD       @_svar+1,#5   ; add 5 to svar.m2
           LRETR                   ; return from function
          .endasmfunc
```

Compile both source files with the −g option and link them as follows:

```
cl2000 −v28 −g cvars.c addfive.asm −z −l=lnk.cmd −l=rts2800.lib −o=addfive.out
```

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

## 3.13 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 4-65). The assembler appends the cross-reference to the end of the source listing.

*Example 3−6. An Assembler Cross-Reference Listing*

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

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 3−3 lists these characters and names.

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

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

*Table 3−3. Symbol Attributes*

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

## 3.14 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 3−4 lists the instruction to be changed and the change that occurs.

*Table 3−4. Smart Encoding for Efficiency*

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

The assembler also intuitively changes instruction formats during smart encoding. For example, to push the accumulator value to the stack, you use MOV *SP++, ACC. Since it would be intuitive to use PUSH ACC for this operation, the assembler accepts PUSH ACC and through smart encoding, changes it to MOV *SP++, ACC. Table 3−5 shows a list of instructions recognized during intuitive smart encoding and what the instruction is changed to.

*Table 3−5. Smart Encoding Intuitively*

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

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

*Table 3−6. Instructions That Avoid Smart Encoding*

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

## 3.15 C-Type Symbolic Debugging for Assembly Variables (−mg option)

When you assemble 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 assembler 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
```

## 3.16 TMS320C28x Assembler Modes

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

| | |
|---|---|
| −v27 | C27x object mode |
| −v28 | C28x object mode |
| −v28 −m27 | C28x object mode − Accept C27x Syntax Mode |
| −v28 −m20 | C28x object mode − Accept C2xlp Syntax Mode |

The options −m27 and −m20 imply −v28 and hence −v28 need not be specified with these options.

When multiple versions are specified, the assembler uses the first version specified and ignores the rest. For example the command ’asm2000 −v28 −v27’ invokes the assembler in the C28x object mode and the assembler ignores the −v27 switch. Also the assembler issues the following warning:

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

Since −m27 and −m20 imply the version −v28 the command ’asm2000 −m20 −v27’ is equivalent to ’asm2000 −v28 −m20 −v27’ and hence the assembler generates the above warning and ignores the −v27 switch.

The −m27 and −m20 switches cannot be combined. The command ’asm2000 −m20 −m27’ generates the following warning:

```
>> Target option –m27 cannot be combined with –m20; –m27 is ignored
```

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

### 3.16.1 C27x Object Mode

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

*Table 3−7. Non-TMS320C27x Instructions Supported in the C27x Object Mode*

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

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

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

### 3.16.2 C28x Object Mode

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

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

### 3.16.3 C28x Object − Accept C27x Syntax Mode

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

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

### 3.16.4 C28x Object − Accept C2xlp Syntax Mode

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

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

Refer to the *TMS320C2xx User's Guide* for more information about C2xlp instructions.

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

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

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

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

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

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

- ❏ The C1x/C2x/C2xx/C5x assembler accepts numbers with leading zero as decimal integers, that is 010 is treated as 10 and not as 8. The C28x assembler treats constants with leading zeros as octal numbers. There may be C2xlp assembly code that contains decimal numbers with leading zeros. When these files are assembled with the C28x assembler the results will not be what you expect as the C28x assembler treats such constants as octal numbers. So the assembler when invoked with −m20 −mw, checks for such numbers and issues a warning that the constant is parsed as an octal number.

  For example, consider the following listing produced using the −m20 −mw options:

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

- The C1x/C2x/C2xx/C5 assembler uses a different order of operator precedence expression. In the C1x/C2x/C2xx/C5 assembler, the shift operators (<< and >>) have higher precedence than the binary + and − operators. The C28x assembler follows the order of precedence of C language where the above mentioned sequence is reversed. The C28x assembler issues a warning about the precedence used if the following are true:
  - The −m20 −mw options are specified.
  - The source code contains any expression involving binary additive operators (+ and −) and the shift operators (<< and >>).
  - The precedence is not forced by parentheses. For example, consider the following listing produced using the −m20 −mw options:

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

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 4.11) describes the directives according to function, and the second part (section 4.12) is an alphabetical reference.

**Topic** **Page**

**4.1 Directives Summary . . . . . . . . . . 4-2**

**4.2 Compatibility With the TMS320C1x/C2x/C2xx/C5x Assembler Directives . . . . . . . . . . 4-7**

**4.3 Directives That Define Sections . . . . . . . . . . 4-8**

**4.4 Directives That Initialize Constants . . . . . . . . . . 4-10**

**4.5 Directive That Aligns the Section Program Counter . . . . . . . . . . 4-13**

**4.6 Directives That Format the Output Listing . . . . . . . . . . 4-15**

**4.7 Directives That Reference Other Files . . . . . . . . . . 4-17**

**4.8 Directives That Enable Conditional Assembly . . . . . . . . . . 4-18**

**4.9 Directives That Define Symbols at Assembly Time . . . . . . . . . . 4-19**

**4.10 Assembler Mode Override Directives . . . . . . . . . . 4-20**

**4.11 Miscellaneous Directives . . . . . . . . . . 4-21**

**4.12 Directives Reference . . . . . . . . . . 4-22**

## 4.1 Directives Summary

In addition to the assembler directives documented here, the TMS320C28x™ software tools support the following directives:

- **Macro directives**. The assembler uses several directives for macros. Macro directives are discussed in Chapter 5, *Macro Language*; they are not discussed in this chapter.
- **Symbolic debugging directives**. The C/C++ compiler uses directives for symbolic debugging. Unlike other directives, symbolic debugging directives are not used in most assembly language programs. Appendix B, *Symbolic Debugging Directives*, discusses these directives; they are not discussed in this chapter.

**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 | 4-71 |
| **.bss** *symbol*, *size in words* [, *blocking*] [, *alignment* ] | Reserves *size* words in the .bss (uninitialized data) section | 4-27 |
| **.data** | Assembles into the .data (initialized data) section | 4-36 |
| **.sect** **”***section name***”** | Assembles into a named (initialized) section | 4-69 |
| **.text** | Assembles into the .text (executable code) section | 4-78 |
| *symbol* **.usect** **”***section name***”,** *size in words* [, *blocking*] [,*alignment flag*] | Reserves *size* words in a named (uninitialized) section | 4-80 |
| **.sblock** | Designates section for blocking | 4-69 |

*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 | 4-30 |
| **.field** *value* [**,** *size*] | Initializes a field of *size* bits (1−32) with *value* | 4-42 |
| **.float** *value*$_1$ [**,** ... **,** *value*$_n$] | Initializes one or more 32-bit IEEE single-precision floating-point constants | 4-45 |
| **.int** *value*$_1$ [**,** ... **,** *value*$_n$] | Initializes one or more 16-bit integers | 4-51 |
| **.long** *value*$_1$ [**,** ... **,** *value*$_n$] | Initializes one or more 32-bit integers | 4-56 |
| **.pstring** | Places 8-bit characters from a character string into the current section | 4-73 |
| **.space** *size* | Reserves *size* bits in the current section; a label points to the beginning of the reserved space | 4-71 |
| **.string** {*expr*$_1$ \| **"***string*$_1$**"**} [**,** ... **,** {*expr*$_n$ \| **"***string*$_n$**"**}] | Initializes one or more text strings | 4-73 |
| **.word** *value*$_1$ [**,** ... **,** *value*$_n$] | Initializes one or more 16-bit integers | 4-51 |
| **.xfloat** | Places the floating-point representation of one or more floating-point constants into the current section | 4-45 |
| **.xlong** | Places on or more 32-bit values into consecutive words in the current section | 4-56 |

*(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 | 4-22 |

*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) | 4-37 |
| **.drnolist** | Suppresses listing of certain directive lines | 4-37 |
| **.fclist** | Allows false conditional code block listing (default) | 4-41 |
| **.fcnolist** | Suppresses false conditional code block listing | 4-41 |
| **.length** *page length* | Sets the page length of the source listing | 4-53 |
| **.list** | Restarts the source listing | 4-54 |
| **.mlist** | Allows macro listings and loop blocks (default) | 4-61 |
| **.mnolist** | Suppresses macro listings and loop blocks | 4-61 |
| **.nolist** | Stops the source listing | 4-54 |
| **.option** *$option_1$* [**,** *$option_2$***,** ...] | Selects output listing options; available options are B, L, M, R, T, W, and X | 4-65 |
| **.page** | Ejects a page in the source listing | 4-67 |
| **.sslist** | Allows expanded substitution symbol listing | 4-72 |
| **.ssnolist** | Suppresses expanded substitution symbol listing (default) | 4-72 |
| **.tab** *size* | Sets tab size | 4-77 |
| **.title "***string***"** | Prints a title in the listing page heading | 4-79 |
| **.width** *page width* | Sets the page width of the source listing to *page width* | 4-53 |

*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 | 4-33 |
| **.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 | 4-46 |
| **.global** *symbol*$_1$ [**,** ... **,** *symbol*$_n$] | Identifies one or more global (external) symbols | 4-46 |
| **.include** [**"**]*filename*[**"**] | Includes source statements from another file | 4-33 |
| **.mlib** [**"**]*filename*[**"**] | Defines macro library | 4-59 |
| **.ref** *symbol*$_1$ [**,** ... **,** *symbol*$_n$] | Identifies one or more symbols used in the current module that are defined in another module | 4-46 |

*(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. | 4-57 |
| **.else** | Assembles code block if the .if *well-defined expression* is false. When using the .if construct, the .else construct is optional. | 4-49 |
| **.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. | 4-49 |
| **.endif** | Ends .if code block | 4-49 |
| **.endloop** | Ends .loop code block | 4-57 |
| **.if** *well-defined expression* | Assembles code block if the *well-defined expression* is true | 4-49 |
| **.loop** [*well-defined expression*] | Begins repeatable assembly of a code block; the loop count is determined by the w*ell-defined expression*. | 4-57 |

*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* | 4-23 |
| **.endstruct** | Ends structure definition | 4-74 |
| **.eval** *well-defined expression*, *substitution symbol* | Performs arithmetic on numeric *substitution symbol* | 4-23 |
| **.label** *symbol* | Defines a load-time relocatable label in a section | 4-52 |
| *symbol* **.set** *value* | Equates *value* with *symbol* | 4-70 |
| **.struct** | Begins structure definition | 4-74 |
| **.tag** | Assigns structure attributes to a label | 4-74 |

*(h) Assembly mode override directives*

| Mnemonic and Syntax | Description | Page |
|---|---|---|
| **.c28_amode** | Begin assembling in C28x object mode. | 4-31 |
| **.lp_amode** | Begin assembling in C28x object mode −− accept C2xLP | 4-31 |

*(i) Miscellaneous directives*

| Mnemonic and Syntax | Description | Page |
|---|---|---|
| **.asmfunc** | Identify the beginning of a block of code that contains a function | 4-25 |
| **.clink** [**"***section name***"**] | Enables conditional linking for the current or specified section. | 4-32 |
| **.emsg** *string* | Sends user-defined error messages to the output device; produces no .obj file | 4-38 |
| **.end** | Ends program | 4-40 |
| **.endasmfunc** | Identify the end of a block of code that contains a function | 4-25 |
| **.mmsg** *string* | Sends user-defined messages to the output device | 4-38 |
| **.newblock** | Undefines local labels | 4-64 |
| **.wmsg** *string* | Sends user-defined warning messages to the output device | 4-38 |

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

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

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

- The .mmregs and .port directives are supported by the C1x/C2x/C2xx/C5x assembler. The C28x assembler when invoked with the −m20 option, ignores these directives and issues a warning that the directives are ignored. The C28x assembler does not accept these directives.

## 4.3 Directives That Define Sections

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

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

Chapter 2, *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.

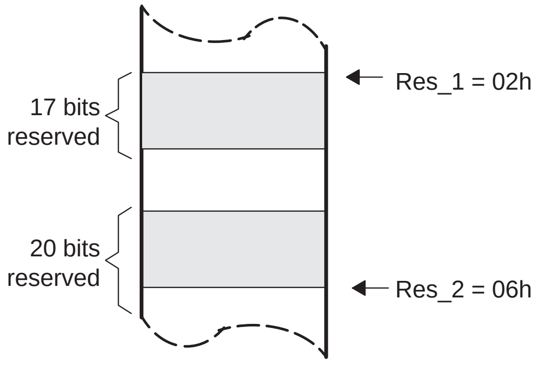
  - When you use a label with .space, it points to the *first* word that contains reserved bits.
  - When you use a label with .bes, it points to the *last* word that contains reserved bits.

  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*



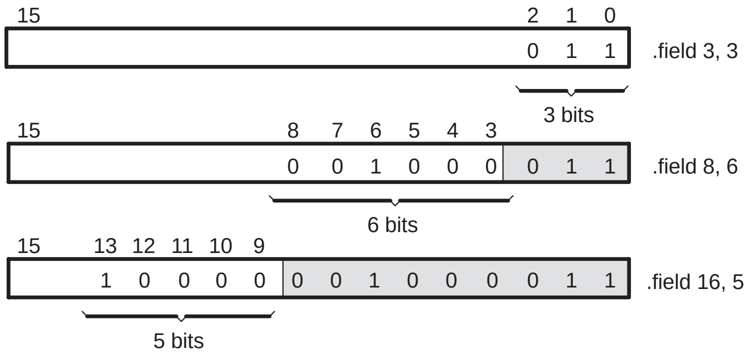
❑ The **.byte** and **.char** directives place one or more 8-bit values into consecutive words of the current section. This directive is similar to .word, except that the width of each value is restricted to eight bits.

❑ 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.

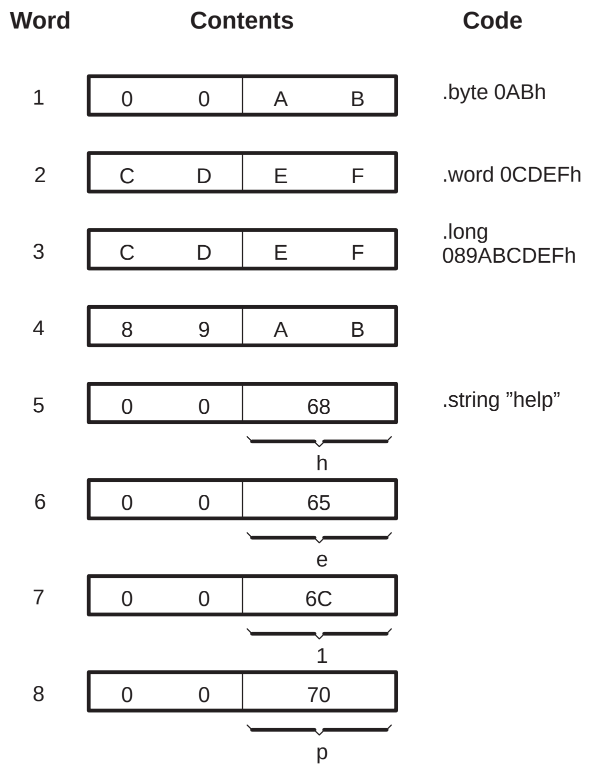
**Note: Directives That Initialize Constants When Used in a .struct/.endstruct Sequence**

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 4-74.

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 |

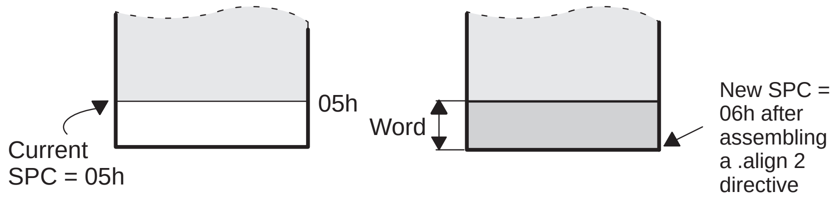
The .align directive with no operands defaults to 64, that is, to a page boundary.

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

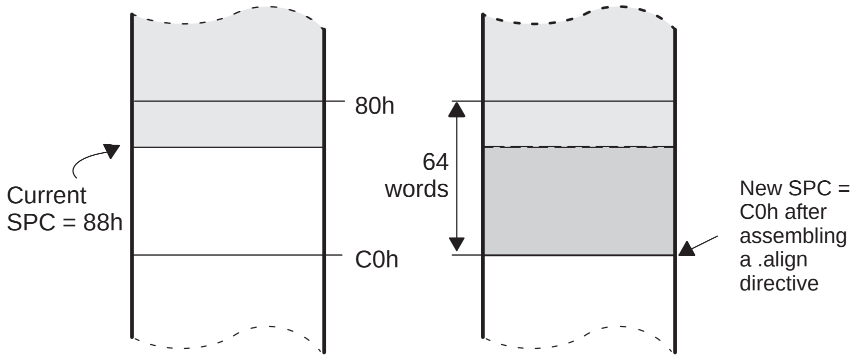
```
1 000000 0002          .field  2,3
2 000000 005A          .field  11,8
3                      .align  2
4 000002 0065          .string ”errorcnt”
  000003 0072
  000004 0072
  000005 006F
  000006 0072
  000007 0063
  000008 006E
  000009 0074
5                      .align
6 000040 0004         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

  | | |
  |---|---|
  | **A** | turns on the listing of all directives, data, and subsequent expansions, macros, and blocks |
  | **B** | limits the listing of the .byte directive to one line. |
  | **D** | turns off the listing of certain directives (performs a .drnolist) |
  | **L** | limits the listing of .long directives to one line. |
  | **M** | turns off macro expansions in the listing. |
  | **N** | turns off listing (performs a .nolist) |
  | **O** | turns on listing (performs a .list) |
  | **R** | resets the B, L, M, T, and W directives (turns off the limits of B, L, M, T, and W). |

| | |
|---|---|
| **T** | limits the listing of .string directives to one line. |
| **W** | limits the listing of .word and .int directives to one line. |
| **X** | produces a cross-reference listing of symbols. You can also obtain a cross-reference listing by invoking the assembler with the −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 3.9.4, *Conditional Expressions*, on page 3-27.

## 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, T_LEN
        ADD     ACC, @COORD.Y

        .bss    COORD
```

## 4.10 Assembler Mode Override Directives

These directives override the global syntax checking modes discussed in section 3.16, *TMS320C28x Assembler Modes*, on page 3-41. These directives are not valid with the C27x Object Mode (−v27 option).

- The .c28_amode directive sets the assembler mode to C28x Object Mode (−v28). The instructions after this directive are assembled in C28x Object Mode regardless of the option used in the command line.
- The .lp_amode directive sets the assembler mode to C28x Object Mode − Accept C2xlp instruction syntax (−m20). The instructions after this directives are assembled as if the −m20 options is specified on the command line.

## 4.11 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler −gw option to generate debug information for separate functions.
- 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 3.8.2, *Local Labels*, on page 3-18.)

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.

## 4.12 Directives Reference

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

### .align

*Align SPC on the Next Boundary*

**Syntax**

**.align** [*size in words*]

**Description**

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size in words* parameter. The *size* 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 byte boundary |
| | 2 | aligns SPC to long word/even boundary |
| | 64 | aligns SPC to page boundary |

Using the .align directive has two effects:

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

**Example**

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

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

## .asg/.eval — *Assign a Substitution Symbol*

**Syntax**

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

**Description**

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

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

❑ The *substitution symbol* 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
       2                        .asg    XAR6, FP
       3 00000000 0964            ADD     ACC, #100
       4 00000001 7786            NOP     *FP++
#                              NOP     *XAR6++
       5 00000002 7786            NOP     *XAR6++
       6
       7                        .asg    0, x
       8                        .loop   5
       9                        .eval   x+1, x
      10                        .word   x
      11                        .endloop
1                               .eval   x+1, x
#                               .eval   0+1, x
1        00000003 0001            .word   x
#                               .word   1
1                               .eval   x+1, x
#                               .eval   1+1, x
1        00000004 0002            .word   x
#                               .word   2
1                               .eval   x+1, x
#                               .eval   2+1, x
1        00000005 0003            .word   x
#                               .word   3
1                               .eval   x+1, x
#                               .eval   3+1, x
1        00000006 0004            .word   x
#                               .word   4
1                               .eval   x+1, x
#                               .eval   4+1, x
1        00000007 0005            .word   x
#                               .word   5
```

## .asmfunc/ .endasmfunc

*Mark Function Boundaries*

**Syntax**

*symbol* **.asmfunc**
**.endasmfunc**

**Description**

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

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

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

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

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

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

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

```
 1 00000000                   .sect        ”.text”
 2                            .global      userfunc
 3                            .global      _printf
 4
 5             userfunc:      .asmfunc
 6 00000000 FE02              ADDB      SP,#2
 7 00000001 2841+             MOV       *–SP[1],#SL1
   00000002 0000
 8 00000003 7640!             LCR       #_printf
   00000004 0000
 9 00000005 9A00              MOVB      AL,#0
10 00000006 FE82              SUBB      SP,#2
11 00000007 0006              LRETR
12                            .endasmfunc
13
14 00000000                   .sect        ”.const”
15 00000000 0048  SL1:        .string      ”Hello World!”,10,0
   00000001 0065
   00000002 006C
   00000003 006C
   00000004 006F
   00000005 0020
   00000006 0057
   00000007 006F
   00000008 0072
   00000009 006C
   0000000a 0064
   0000000b 0021
   0000000c 000A
   0000000d 0000
```

## .bss — *Reserve Space in the .bss Section*

**Syntax**

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

**Description**

The **.bss** directive reserves space for variables in the .bss section. This directive is usually used to allocate 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 3.15, *C-Type Symbolic Debugging for Assembly Variables (−mg option)*, on page 3-39 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.

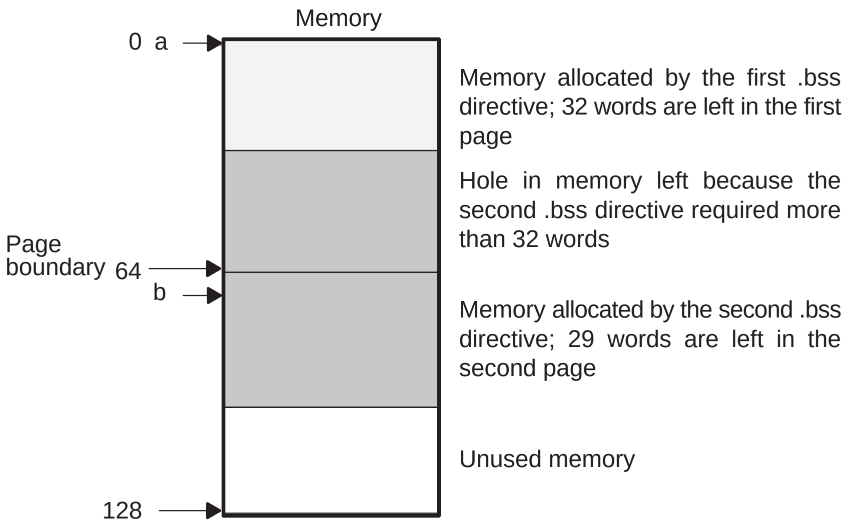
- ❑ If you do not request blocking, the memory is allocated at the current SPC.
- ❑ If you request blocking, the assembler checks to see whether there is enough space between the current SPC and the page boundary. If there is not enough space, the assembler creates another hole and allocates the space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


Memory
0 a
Memory allocated by the first .bss directive; 32 words are left in the first page
Hole in memory left because the second .bss directive required more than 32 words
Page boundary 64
b
Memory allocated by the second .bss directive; 29 words are left in the second page
Unused memory
128

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
```

## .byte/.char *Initialize Byte*

**Syntax**

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

**Description**

The **.byte** and **.char** directives place one or more 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
3 000106 000A        .char   10, −1, “abc”, ’a’
  000107 00FF
  000108 0061
  000109 0062
  00010a 0063
  00010b 0061
```

## .c28_amode/ .lp_amode

*Override Assembler Mode*

**Syntax**

**.c28_amode**
**.lp_amode**

**Description**

The **.c28_amode** and **.lp_amode** directives tell the assembler to override the assembler mode. See section 3.16, *TMS320C28x Assembler Modes*, on page 3-41, for more information.

The .c28_amode directive tells the assembler to operate in the C28x object mode (−v28). The .lp_amode directive tells the assembler to operate in C28x object − accept C2xlp syntax mode (−m20). These directives can be repeated throughout a source file.

For example, if a file is assembled with the −m20 option, the assembler begins the assembly in the C28x object − accept C2xlp syntax mode. When it encounters the .c28_amode directive, it changes the mode to C28x object mode and remains in that mode until it encounters a .lp_amode directive or the end of file.

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

**Example**

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

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

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

## .clink — *Conditionally Leave Section Out of COFF Output*

**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 8500+          MOV     ACC, @X
21 000002 3100           MOV     P, #0
22 000003 oFAB           CMPL    ACC, P
```

## .copy/.include — *Copy Source File*

**Syntax**

**.copy** [”]*filename*[”]
**.include** [”]*filename*[”]

**Description**

The **.copy** and **.include** directives tell the assembler to read source statements from 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 quotes and must follow operating system conventions. If *filename* starts with a number the double quotes are required.

You can specify a full pathname (for example, 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 variables C2000_A_DIR and A_DIR

For more information about the −i option, and C2000_A_DIR and A_DIR, see section 3.4, *Naming Alternate Directories for Assembler Input*, on page 3-7.

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` | `** In byte.asm` | `** In word.asm` |
| **`.copy ”byte.asm”`** | `.byte 32,1+ ’A’` | `.word 0ABCDh, 56q` |
| | **`.copy ”word.asm”`** | |
| `**Back in original file` | `** Back in byte.asm` | |
| `.pstring ”done”` | `.byte 67h + 3q` | |

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

**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
```

## .data — *Assemble Into .data Section*

**Syntax** **.data**

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

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.

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

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

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

## .drlist/.drnolist *Control Listing of Directives*

**Syntax**

**.drlist**
**.drnolist**

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

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

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

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

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

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

**Source file:**

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

     .drnolist

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

**Listing file:**

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

## .emsg/.mmsg/ .wmsg

*Define Messages*

**Syntax**

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

**Description**

These directives allow you to define your own error and warning messages. 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
```

## .end — *End Assembly*

**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
```

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

**Syntax**

**.fclist**
**.fcnolist**

**Description**

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

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

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

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

**Example**

This example shows the assembly language and listing files for code with and without the 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
```

## .field — *Initialize Field*

**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 4-74.

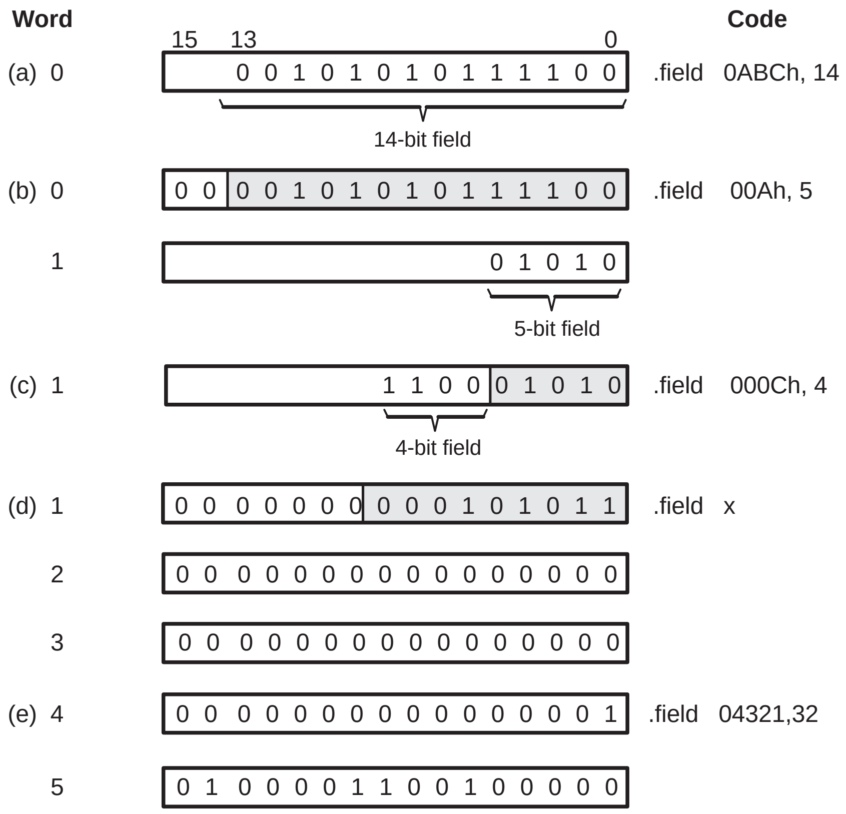
**Example**

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*



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

**Syntax**

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

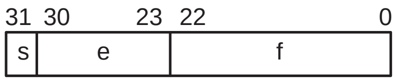
**Description**

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 and .xfloat directives.

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

## .global Identify Global Symbols

**Syntax**

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

**Description**

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

The **.global** directive is provided for backward compatibility for C2xlp source. It is accepted only when the −m20 option is specified. The use of .globl is discouraged.

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
```

## .if/.elseif/.else/ .endif

*Assemble Conditional Blocks*

**Syntax**

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

**Description**

The following directives provide conditional assembly:

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

- If the expression evaluates to *true* (nonzero), the assembler assembles the code that follows the expression (up to a .elseif, .else, or .endif).
- If the expression evaluates to *false* (0), the assembler assembles 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). The .else directive is optional in the conditional block; if an expression is false and there is no .else statement, the assembler continues with the code that follows the .endif.

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

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

For information about relational operators, see section 3.9.4, *Conditional Expressions*, on page 3-27.

**Example** This example shows conditional assembly.

```
1         0001  SYM1   .set    1
2         0002  SYM2   .set    2
3         0003  SYM3   .set    3
4         0004  SYM4   .set    4
5
6               If_4:  .if     SYM4 = SYM2 * SYM2
7 000000 0004          .byte   SYM4       ; Equal values
8                      .else
9                      .byte   SYM2 * SYM2 ; Unequal values
10                     .endif
11
12               If_5:  .if     SYM1 <= 10
13 000001 000A          .byte   10       ; Less than / equal
14                      .else
15                      .byte   SYM1     ; Greater than
16                      .endif
17
18               If_6:  .if     SYM3 * SYM2 != SYM4 + SYM2
19                      .byte   SYM3 * SYM2  ; Unequal value
20                      .else
21 000002 0008          .byte   SYM4 + SYM4  ; Equal values
22                      .endif
23
24               If_7:  .if SYM1 = 2
25                      .byte   SYM1
26                      .elseif SYM2 + SYM3 = 5
27 000003 0005          .byte   SYM2 + SYM3
28                      .endif
```

## .int/.word — *Initialize 16-Bit Integer*

**Syntax**

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

**Description**

The **.int** and **.word** directives 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 Tim*e, 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
```

## .label — *Create a Load−Time Address Label*

**Syntax** **.label** *symbol*

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

For some applications, it is desirable to have a section load at one address and run at a different address. For example, you may 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 run-time address so that references to the section (such as branches) are correct when the code runs.

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

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

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

For more information about assigning run-time and load-time addresses in the linker, see section 7.9, *Specifying a Section’s Run-Time Address*, on page 7-39.

## .length/.width — *Set Listing Page Size*

**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
```

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

**Syntax**

**.list**
**.nolist**

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

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

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

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

By default, the source listing is printed to the listing file; the assembler acts as if the .list directive had been 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) | copy2.asm (copy file) |
|---|---|
| <pre> .copy   ”copy2.asm”<br>* Back in original file<br> NOP<br> **.nolist**<br> .copy   ”copy2.asm”<br> **.list**<br>* Back in original file<br> .string ”Done”</pre> | <pre>*In copy2.asm (copy file)<br>      .word 32, 1 + ’A’</pre> |

**Listing file:**

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

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

**Syntax**

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

**Description**

The **.long** and **.xlong** directives place one or more 32-bit values into consecutive words in the current section. The most significant word is stored first. The .long directive aligns the result on the long word boundary, while 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:
```

## .loop/.break/ .endoop

*Assemble Code Block Repeatedly*

**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
```

## .macro/.endm *Define Macro*

**Syntax**

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

**Description**

The **.macro** directive is used to define macros.

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

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

Macros are explained in further detail in Chapter 5, *Macro Language*.

## .mlib — *Define Macro Library*

**Syntax**

**.mlib** [**”**]*filename*[**”**]

**Description**

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

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

1) The directory that contains the current source file
2) Any directories named with the −i assembler option
3) Any directories specified by the C2000_A_DIR or A_DIR environment variable

For more information about the −i option and the environment variable, see section 3.4, *Naming Alternate Directories for Assembler Input*, on page 3-7.

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

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

**Example**

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

| inc1.asm | dec1.asm |
|---|---|
| `* Macro for incrementing`<br>`inc1  .macro    A`<br>`      ADD       A, #1`<br>`      .endm` | `* Macro for decrementing`<br>`dec1  .macro    A`<br>`      SUB       A, #1`<br>`      .endm` |

Use the archiver to create a macro library:

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

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

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

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

**Syntax**

**.mlist**
**.mnolist**

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

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

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

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

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

```
       1              STR_3  .macro   P1, P2, P3
       2                     .string ":p1:", ":p2:", ":p3:"
       3                     .endm
       4
       5 000000               STR_3 "as", "I", "am"
1        000000 003A         .string ":p1:", ":p2:", ":p3:"
         000001 0070
         000002 0031
         000003 003A
         000004 003A
         000005 0070
         000006 0032
         000007 003A
         000008 003A
         000009 0070
         00000a 0033
         00000b 003A
       6 00000c 003A         .string ":p1:", ":p2:", ":p3:"
         00000d 0070
         00000e 0031
         00000f 003A
         000010 003A
         000011 0070
         000012 0032
         000013 003A
         000014 003A
         000015 0070
         000016 0033
         000017 003A
       7
       8                     .mnolist
       9 000018               STR_3 "as", "I", "am"
      10                     .mlist
      11 000024               STR_3 "as", "I", "am"
1        000024 003A         .string ":p1:", ":p2:", ":p3:"
         000025 0070
         000026 0031
         000027 003A
         000028 003A
         000029 0070
         00002a 0032
         00002b 003A
         00002c 003A
         00002d 0070
         00002e 0033
         00002f 003A
```

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

## .newblock — *Terminate Local Symbol Block*

**Syntax** **.newblock**

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

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 3.8.2 on page 3-18.

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

```
 1                     .ref    ADDRA, ADDRB, ADDRC
 2         0076  B       .set    76h
 3
 4 00000000 F800!          MOV     DP, #ADDRA
 5
 6 00000001 8500! LABEL1: MOV     ACC, @ADDRA
 7 00000002 1976           SUB     ACC, #B
 8 00000003 6403           B       $1, LT
 9 00000004 9600!          MOV     @ADDRB, ACC
10 00000005 6F02           B       $2, UNC
11
12 00000006 8500! $1       MOV     ACC, @ADDRA
13 00000007 8100! $2       ADD     ACC, @ADDRC
14                       .newblock       ; Undefine $1 to use again.
15
16 00000008 6402           B       $1, LT
17 00000009 9600!          MOV     @ADDRC, ACC
18 0000000a 7700  $1       NOP
```

## .option — *Select Listing Options*

**Syntax** **.option** *option 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:

| | |
|---|---|
| **A** | turns on the listing of all directives, data, and subsequent expansions, macros, and blocks |
| **B** | limits the listing of .byte directives to one line. |
| **D** | turns off the listing of certain directives (performs a .drnolist) |
| **L** | limits the listing of .long directives to one line. |
| **M** | turns off macro expansions in the listing. |
| **N** | turns off listing (performs a .nolist) |
| **O** | turns on listing (performs a .list) |
| **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
```

## .page — *Eject Page in Listing*

**Syntax** **.page**

**Description** The **.page** directive produces a page eject in the listing file. The .page directive is not printed in the source listing, but the assembler increments the line counter when it encounters 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            ;        .
TMS320C2000 COFF Assembler       Version x.xx     Day      Time      Year
 Copyright (c) xxxx-xxxx    Texas Instruments Incorporated

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

## .sblock *Specify Blocking for an Initialized Section*

**Syntax**

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

**Description**

The **.sblock** directive designates sections for blocking. Blocking is an address alignment mechanism similar to page alignment, but weaker. A blocked section does not cross a page boundary (64 words) if it is smaller than a page, and it starts on a page boundary if it is larger than a page. This directive allows specification of blocking for initialized sections only, not for uninitialized sections declared with .usect or the .bss directives. The *section names* may optionally be enclosed in quotation marks.

**Example**

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

```
****************************************
** Specify blocking for the .text     **
** and .data sections.                **
****************************************
          .sblock       .text, .data
```

## .sect — *Assemble Into Named Section*

**Syntax**

**.sect** ”*section name*”

**Description**

The **.sect** directive defines a named section that can be used like the default .text and .data sections. The .sect directive 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
```

## .set — *Define Assembly-Time Constant*

**Syntax** *symbol* **.set** *value*

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

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

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

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

**Example** This example shows how symbols can be assigned with .set.

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

## .space/.bes *Reserve Space*

**Syntax**

**.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  **
18             **         contains reserved bits.        **
20             ********************************************
21 000008         RES_1: .space  100
22 00000f 000F           .word   15
23             ********************************************
24             **  Reserve 20 bits in the .data section;  **
25             **  RES_2 points to the last word that    **
26             **         contains reserved bits.        **
27             ********************************************
28 000011         RES_2: .bes    20
29 000012 0036           .word   36h
30 000013 0011”          .word   RES_
```

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

**Syntax**

**.sslist**
**.ssnolist**

**Description**

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

## .string/.pstring — *Initialize Text*

**Syntax**

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

**Description**

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

- ❑ 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
```

## .struct/ .endstruct/.tag

*Declare Structure Type*

**Syntax**

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

**Description**

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

**Note: The .struct Directive Does Not Allocate Memory**

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.

- The *element* is one of the following descriptors: .string, .byte, .char, .int, .half, .short, .word, .long, .double, .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 stag must be used (as in the definition of stag).
- The *expr* is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.
- The $expr_{n/N}$ is an optional expression for the number of elements described. This value defaults to 1. A .string element is considered to be one byte in size, and a .field element is one bit.

- The *mem*$_{n/N}$ is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.
- The *size* is an optional label for the total size of the structure.
- The *stag* is the structure's tag. Its value is associated with the beginning of the structure. If no stag is present, the assembler puts the structure members in the global symbol table with the value of their absolute offset from the top of the structure. A .stag is optional for .struct, but is required for .tag.

**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
```

## .tab — *Define Tab Size*

**Syntax**

**.tab** *size*

**Description**

The **.tab** directive defines the tab size. Tabs encountered in the source input are translated to *size* 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
```

## .text — *Assemble Into .text Section*

**Syntax** **.text**

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

The .text section is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you 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.

```
                  ******************************************
                  ** Begin assembling into .data section. **
                  ******************************************
   000000                   .data
   000000 000A              .byte   0Ah, 0Bh
   000001 000B

                  ******************************************
                  ** Begin assembling into .text section. **
                  ******************************************
   000000                   .text
   000000 0041  START:      .string ”A”, ”B”, ”C”
   000001 0042
   000002 0043
   000003 0058  END:        .string ”X”, ”Y”, ”Z”
   000004 0059
   000005 005A

   000006 8100’             ADD     ACC, @START
   000007 8103’             ADD     ACC, @END

                  ******************************************
                  ** Resume assembling into .data section.**
                  ******************************************
   000002                   .data
   000002 000C              .byte   0Ch, 0Dh
   000003 000D

                  ******************************************
                  ** Resume assembling into .text section.**
                  ******************************************
   000008                   .text
   000008 0051              .string ”Quit”
   000009 0075
   00000a 0069
   00000b 0074
```

## .title — *Define Page Title*

**Syntax** **.title** **”***string***”**

**Description** The **.title** directive supplies a title that is printed in the heading on each listing page. The source statement itself is not printed, but the line counter is incremented.

The *string* is a 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:**

```
TMS320C2000 COFF Assembler       Version x.xx            Day      Time      Year
Copyright (c) xxxx-xxxx        Texas Instruments Incorporated

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

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

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

## .usect — *Reserve Uninitialized Space*

**Syntax**

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

**Description**

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

- ❑ The *symbol* points to the first location reserved by this invocation of the .usect directive. The symbol corresponds to the name of the variable for which you are reserving space.
- ❑ The *section name* is significant to 200 characters and must be enclosed in double quotes. This parameter names the uninitialized section. A section name can contain a subsection name in the form *section name*:*subsection name*.
- ❑ The *size in words* is an expression that defines the number of words that are reserved in *section name.*
- ❑ The *blocking flag* is an optional parameter. If specified and nonzero, the flag means that this object will be blocked. Blocking is an address mechanism similar to alignment, but weaker. It means that an object 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. If any object within a section is blocked, then the section is also blocked in that the section will start on a page boundary if it does not fit completely within a page. Note that blocking an object within a section does not guarantee that other objects within the section are blocked. The other objects should similarly be declared with the blocking flag if they need to be blocked.
- ❑ The *alignment flag* is an optional parameter that causes the assembler to allocate size on long word boundaries.
- ❑ The *type* is an optional parameter that causes the assembler to produce the appropriate debug information for the *symbol*. See section 3.15, *C-Type Symbolic Debugging for Assembly Variables (−mg option*), on page 3-39 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*.

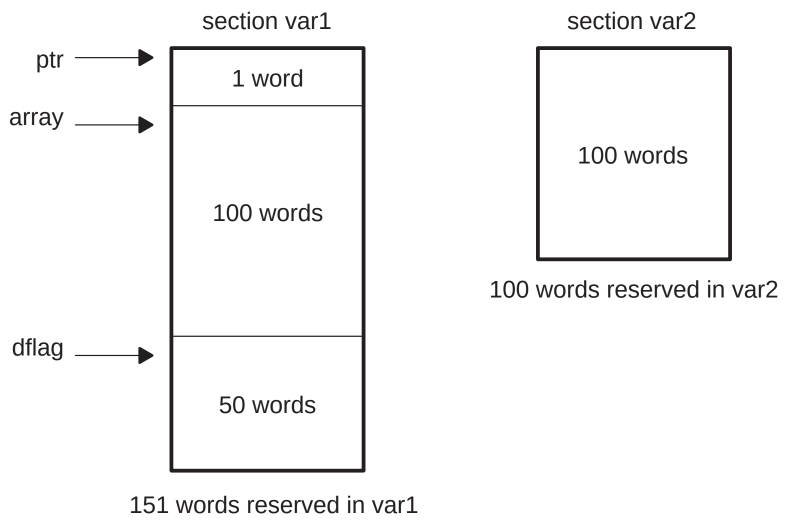
For more information about COFF sections, see Chapter 2, *Introduction to Common Object File Format*.

**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.

```
                  *******************************************
                  **    Assemble into .text section.       **
                  *******************************************
000000                  .text
000000 9A03             MOV     AL, #03h

                  *******************************************
                  **    Reserve 1 word in var1.            **
                  *******************************************
000000              ptr  .usect  ”var1”, 1

                  *******************************************
                  **    Reserve 100 words in var1.         **
                  *******************************************
000001            array  .usect  ”var1”, 100

000001 9C03             ADD     AL, #03h ; Still in .text

                  *******************************************
                  **    Reserve 50 words in var1.          **
                  *******************************************
000065            dflag  .usect  ”var1”, 50

000002 08A9             ADD     AL, #dflag  ; Still in .text
000003 0065-

                  *******************************************
                  **    Reserve 100 words in var2.         **
                  *******************************************
000000              vec  .usect  ”var2”, 100

000004 08A9             ADD     AL, #vec  ; Still in .text
000005 0000-

                  *******************************************
                  **    Declare an external .usect symbol  **
                  *******************************************
                        .global array
```

Figure 4−7 on page 4-82 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 . . . . . . . . . . 5-2**
**5.2 Defining Macros . . . . . . . . . . 5-3**
**5.3 Macro Parameters/Substitution Symbols . . . . . . . . . . 5-5**
**5.4 Macro Libraries . . . . . . . . . . 5-13**
**5.5 Using Conditional Assembly in Macros . . . . . . . . . . 5-14**
**5.6 Using Labels in Macros . . . . . . . . . . 5-16**
**5.7 Producing Messages in Macros . . . . . . . . . . 5-17**
**5.8 Using Directives to Format the Output Listing . . . . . . . . . . 5-19**
**5.9 Using Recursive and Nested Macros . . . . . . . . . . 5-21**
**5.10 Macro Directives Summary . . . . . . . . . . 5-23**

## 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 4-33); 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 3.8.6, *Substitution Symbols*, page 3-24).

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 4-23.

### 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* |
| **$isdefed***(a)* | 1 if string *a* is defined in the symbol table<br>0 if string *a* is not defined in the symbol table |
| **$ismember***(a,b)* | Top member of list *b* is assigned to string *a*<br>0 if *b* is a null string |
| **$iscons***(a)* | 1 if string *a* is a binary constant<br>2 if string *a* is an octal constant<br>3 if string *a* is a hexadecimal constant<br>4 if string *a* is a character constant<br>5 if string *a* is a decimal constant |
| **$isname***(a)* | 1 if string *a* is a valid symbol name<br>0 if string *a* is not a valid symbol name |
| **$isreg***(a)*† | 1 if string *a* is a valid predefined register name<br>0 if string *a* is not a valid predefined register name |

† For more information about predefined register names, see section 3.8.5, *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 4-59). 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 4-61. 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 4-49.

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 4-57.

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 Block*

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

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

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

        .endm

    .global A0, A1, A2

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

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 4-38.

## 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*

```
       .fcnolist

fact   .macro N, loc

       .if N < 2
       MOV   @LOC, #1
       .else
       MOV   @LOC, #N


       .eval N−1, N
fact1  temp

       .endif
       .endm

fact1  .macro
       .if N < 1
       MOV      @T, @LOC
       MPYB     @P, @T, #N
       MOV      @LOC, @P
       MOV      ACC, @LOC
       .eval    N − 1, N
fact1

       .endif
       .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. See the page numbers listed in the tables for details on each directive.

*Table 5−2. Creating Macros*

| Mnemonic and Syntax | Description | Macro Use | 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 | 4-59 |

*Table 5−3. Manipulating Substitution Symbols*

| Mnemonic and Syntax | Description | Macro Use | Directive Description |
|---|---|---|---|
| **.asg** [**"**]*character string*[**"**]*, substitution symbol* | Assign character string to substitution symbol | 5-6 | 4-23 |
| **.eval** *well-defined expression, substitution symbol* | Perform arithmetic on numeric substitution symbols | 5-7 | 4-23 |
| **.var** *sym*$_1$ [,*sym*$_2$, ... ,*sym*$_n$] | Define local macro symbols | 5-12 | 5-12 |

*Table 5−4. Conditional Assembly*

| Mnemonic and Syntax | Description | Macro Use | Directive Description |
|---|---|---|---|
| **.break** [*well-defined expression*] | Optional repeatable block assembly | 5-14 | 4-57 |
| **.endif** | End conditional assembly | 5-14 | 4-49 |
| **.endloop** | End repeatable block assembly | 5-14 | 4-57 |
| **.else** | Optional conditional assembly block | 5-14 | 4-49 |
| **.elseif** *well-defined expression* | Optional conditional assembly block | 5-14 | 4-49 |
| **.if** *well-defined expression* | Begin conditional assembly | 5-14 | 4-49 |
| **.loop** [*well-defined expression*] | Begin repeatable block assembly | 5-14 | 4-57 |

*Table 5−5. Producing Assembly-Time Messages*

| Mnemonic and Syntax | Description | Macro Use | Directive Description |
|---|---|---|---|
| **.emsg** | Send error message to standard output | 5-17 | 4-38 |
| **.mmsg** | Send assembly-time message to standard output | 5-17 | 4-38 |
| **.wmsg** | Send warning message to standard output | 5-17 | 4-38 |

*Table 5−6. Formatting the Listing*

| Mnemonic and Syntax | Description | Macro Use | Directive Description |
|---|---|---|---|
| **.fclist** | Allow false conditional code block listing (default) | 5-19 | 4-41 |
| **.fcnolist** | Suppress false conditional code block listing | 5-19 | 4-41 |
| **.mlist** | Allow macro listings (default) | 5-19 | 4-61 |
| **.mnolist** | Suppress macro listings | 5-19 | 4-61 |
| **.sslist** | Allow expanded substitution symbol listing | 5-19 | 4-72 |
| **.ssnolist** | Suppress expanded substitution symbol listing (default) | 5-19 | 4-72 |

Chapter 6

# Archiver Description

The TMS320C28x™ archiver combines 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 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-2**
**6.2 Archiver Development Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-3**
**6.3 Invoking the Archiver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-4**
**6.4 Archiver Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-6**

## 6.1 Archiver Overview

The TMS320C28x archiver lets you combine several individual files into a single file called an archive or a library. Each file within the archive is called a member. Once you have created an archive, you can use the archiver to add, delete, or extract members.

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

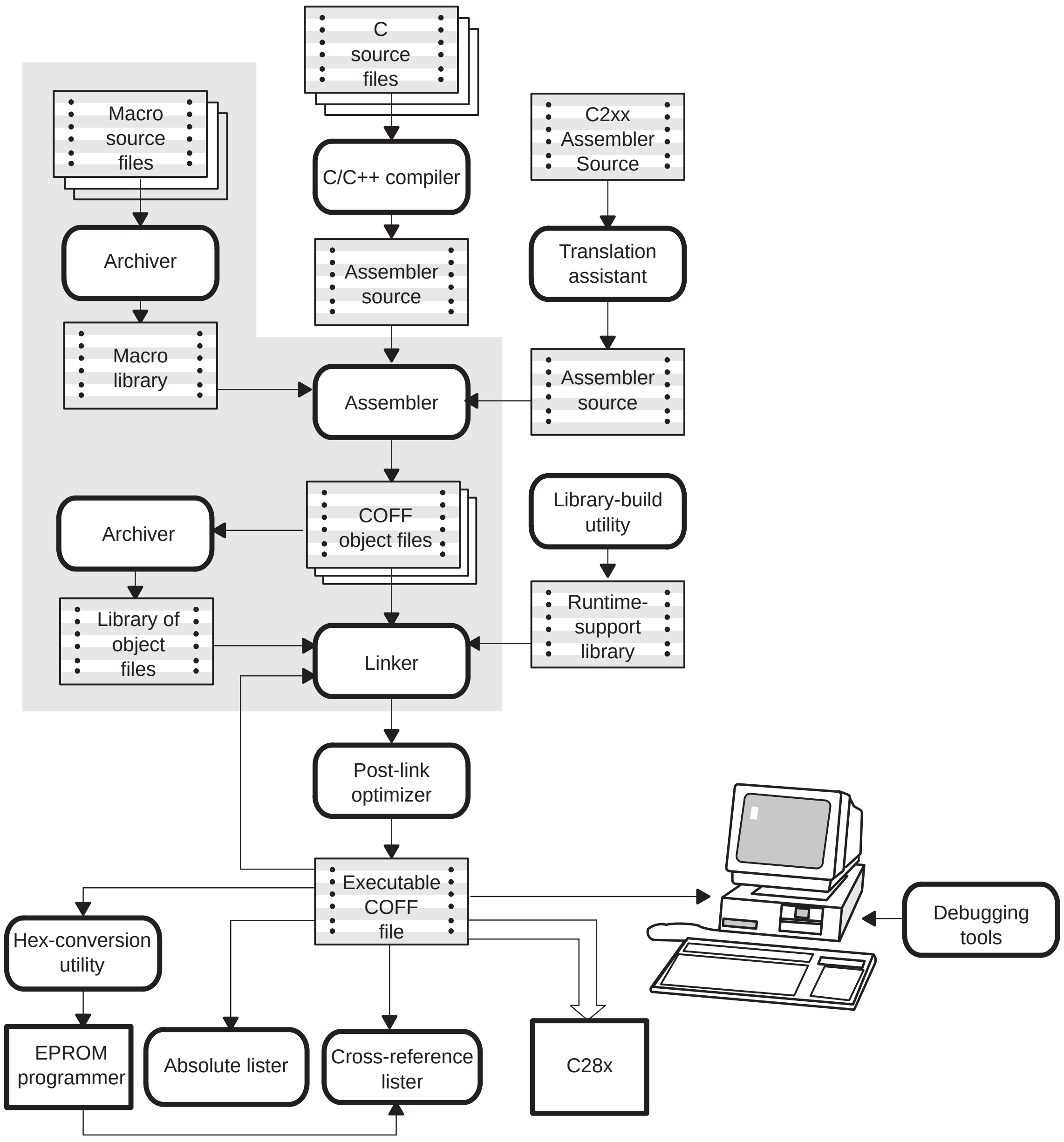
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The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

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

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



## 6.3 Invoking the Archiver

To invoke the archiver, enter:

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

| | |
|---|---|
| **ar2000** | is the command that invokes the archiver. |
| [−]*command* | tells the archiver how to manipulate the existing library members and any specified *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:

  ```
  TMS320C28x 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:

  ```
  TMS320C28x Archiver            Version x.xx
  Copyright (c) xxxx-xxxx Texas Instruments Incorporated

           FILE NAME    SIZE   DATE
    ----------------   -----   ------------------------
            sine.obj     300   Wed Apr 4 10:00:24 2001
             cos.obj     300   Wed Apr 4 10:00:30 2001
             flt.obj     300   Wed Apr 4 09:59:56 2001
  ```

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

  ```
  ar2000 -as function atan.obj
  ```

  The archiver responds as follows:

  ```
  TMS320C28x 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:

  ```
  TMS320C28x 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 TMS320C28x™ 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 . . . . . 7-2**

**7.2 The Linker's Role in the Software Development Flow . . . . . 7-3**

**7.3 Invoking the Linker . . . . . 7-4**

**7.4 Linker Options . . . . . 7-5**

**7.5 Linker Command Files . . . . . 7-19**

**7.6 Object Libraries . . . . . 7-22**

**7.7 The MEMORY Directive . . . . . 7-24**

**7.8 The SECTIONS Directive . . . . . 7-29**

**7.9 Specifying a Section's Run-Time Address . . . . . 7-39**

**7.10 Using UNION and GROUP Statements . . . . . 7-43**

**7.11 Overlaying Pages . . . . . 7-46**

**7.12 Special Section Types (DSECT, COPY, and NOLOAD) . . . . . 7-50**

**7.13 Default Allocation . . . . . 7-51**

**7.14 Assigning Symbols at Link Time . . . . . 7-53**

**7.15 Creating and Filling Holes . . . . . 7-61**

**7.16 Linker-Generated Copy Tables . . . . . 7-65**

**7.17 Partial (Incremental) Linking . . . . . 7-80**

**7.18 Linking C Code . . . . . 7-82**

**7.19 Linker Examples . . . . . 7-86**

## 7.1 Linker Overview

The TMS320C28x 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

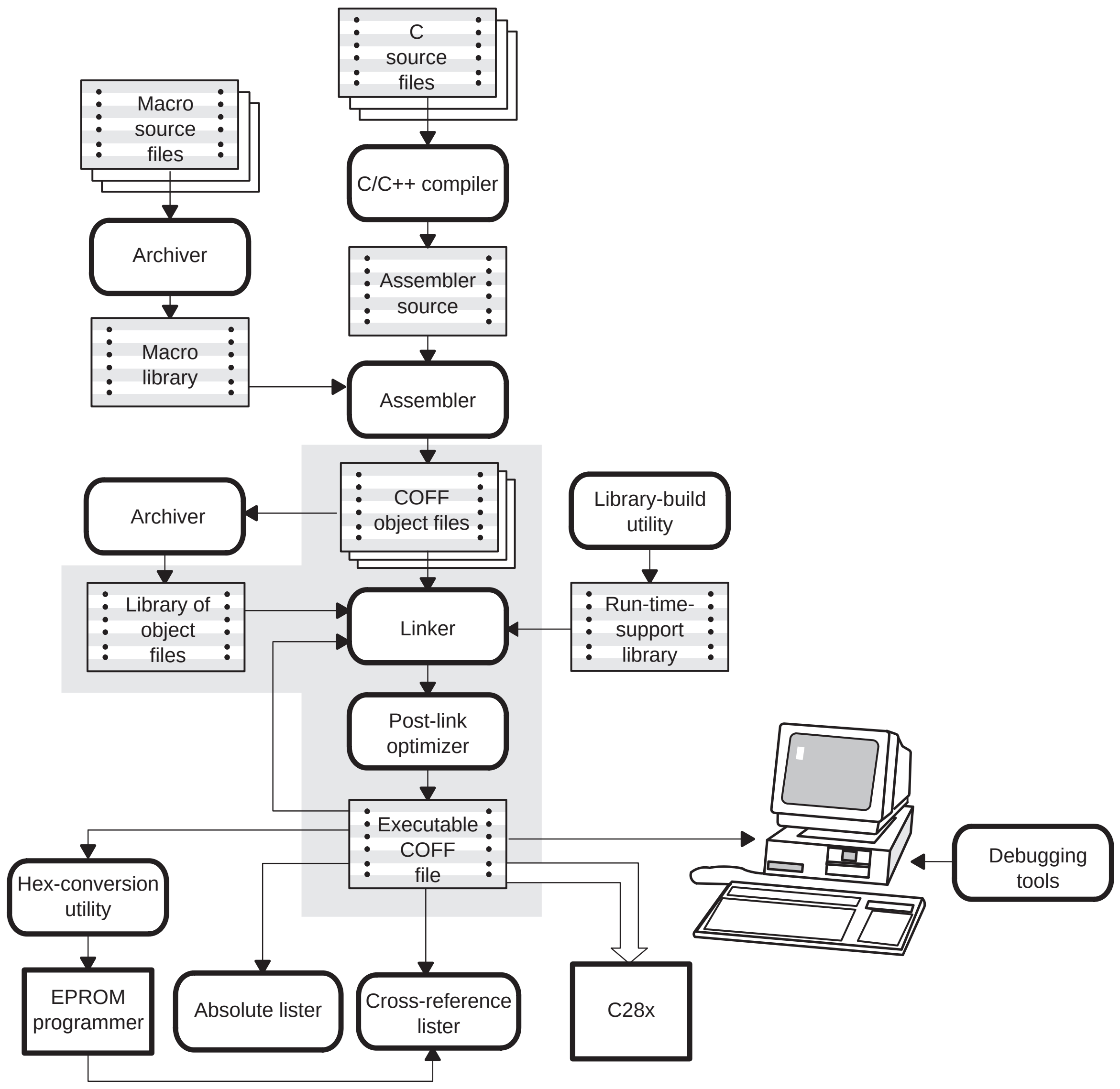
The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow 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 TMS320C28x device.

*Figure 7−1. The Linker in the TMS320C28x Software Development Flow*



## 7.3 Invoking the Linker

The general syntax for invoking the linker is:

**cl2000 −v28 −z** [*options*] *filename*$_1$ , ... [*filename*$_n$]

| | |
|---|---|
| **cl2000 −v28 −z** | is the command that invokes the C28x 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 two methods for invoking the linker:

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

```
cl2000 −v28 −z file1.obj file2.obj −o link.out
```

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

```
−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:

```
cl2000 −v28 −z linker.cmd
```

When you use a command file, you can also specify other options and files on the command line. For example, you could enter:

```
cl2000 −v28 −z −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.

For information on invoking the linker for C/C++ files, see section 7.18, *Linking C/C++ Code*, on page 7-82.

## 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.

*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. | 7-6 |
| **−ar** | Produces a relocatable, executable object module | 7-6 |
| **−b** | Disables merge of symbolic debugging information | 7-8 |
| **−c** | Autoinitializes variables at run time | 7-8 |
| **−cr** | Autoinitializes variables at load time | 7-8 |
| **−e** *global_symbol* | Defines a global *symbol* that specifies the primary entry point for the output module | 7-9 |
| **−f** *fill_value* | Sets the default fill value for holes within output sections; *fill_value* is a 16-bit constant | 7-9 |
| **−g** *symbol* | Makes *symbol* global (overrides −h) | 7-9 |
| **−h** | Makes all global symbols static | 7-10 |
| **−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 | 7-10 |
| **`−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 | 7-11 |
| **−j** | Disables conditional linking | 7-13 |
| **−k** | Forces the linker to ignore any SECTIONS directive alignment specification | 7-13 |
| **−l** *filename* † | Names an archive library or linker command file *filename* as linker input | 7-10 |
| **−m** *filename* † | Produces a map or listing of the input and output sections, including holes, and places the listing in *filename* | 7-13 |

† The *pathname* or *filename* must follow operating-system conventions.
‡ For more information, see the *TMS320C28x Optimizing C/C++ Compiler User's Guide.*

*Table 7−1. Linker Options Summary (Continued)*

| Option | Description | Page |
|---|---|---|
| **−o** *filename* † | Names the executable, output module. The default filename is a.out | 7-14 |
| **−priority** | Satisfies unresolved references by the first library that contains a definition for that symbol | 7-17 |
| **−q** | Suppresses the banner and all progress information (quiet) | 7-15 |
| **−r** | Produces a nonexecutable, relocatable output module | 7-6 |
| **−s** | Strips symbol-table information and line-number entries from the output module | 7-15 |
| **−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 | 7-15 |
| **−u** *symbol* | Places an unresolved external *symbol* into the output module's symbol table | 7-16 |
| **−w** | Displays a message when an undefined output section is created | 7-16 |
| **−x** | Forces rereading of libraries, which resolves back references | 7-17 |
| **−−xml_link_info** *file* | Generates a well-formed XML *file* containing detailed information about the result of a link. | 7-18 |

† The *pathname* or *filename* must follow operating-system conventions.
‡ For more information, see the *TMS320C28x 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 7-56)
- An optional header that describes information such as the program entry point
- *No* unresolved references

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

```
cl2000 −v28 −z −a file1.obj file2.obj
```

**Note: The −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:

  ```
  cl2000 −v28 −z -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 7.17, *Partial (Incremental) Linking*, on page 7-80.)

- ❑ **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:

  ```
  cl2000 −v28 −z -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 7.18, *Linking C Code*, on page 7-82; section 7.18.4, *Autoinitialization of Variables at Run Time*, on page 7-83; and section 7.18.5, *Autoinitialization of Variables at Load Time*, on page 7-84.

### 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.

```
cl2000 −v28 −z −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:

```
cl2000 −v28 −z −f 0ABCDh file1.obj file2.obj
```

For more information on creating and filling holes, see section 7.15, *Creating and Filling Holes*, on page 7-61.

### 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.

```
cl2000 −v28 −z −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:

```
cl2000 −v28 −z −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 7.18, *Linking C Code*, on page 7-82.

### 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:

```
cl2000 −v28 −z 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 up to 128 `−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:

| | |
|---|---|
| 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 |
|---|---|
| UNIX | **`cl2000 −v28 −z f1.obj f2.obj −I/ld −I/ld2 −lr.lib −llib2.lib`** |
| Windows | **`cl2000 −v28 −z 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 |
|---|---|
| UNIX (Bourne shell) | **C_DIR="***pathname*$_1$[; *pathname*$_2$; . . .**";** `export C_DIR` |
| 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:

| | |
|---|---|
| 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 |
|---|---|
| UNIX | `setenv C_DIR "/ld ;/ld2"; export C_DIR`<br>`cl2000 -v28 -z f1.obj f2.obj -l r.lib -l lib2.lib` |
| Windows | `set C_DIR=\ld;\ld2`<br>`cl2000 -v28 -z 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 |
|---|---|
| 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 7-22.

### 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 4-32 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 4-22.

### 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 7-24.

- 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 7-29.

- 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:

```
cl2000 −v28 −z file1.obj file2.obj -m map.out
```

Example 7−23 on page 7-88 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:

```
cl2000 −v28 −z -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.

```
cl2000 −v28 −z −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:

```
cl2000 −v28 −z -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 rts2800.lib contains a member that defines the symbol symtab; none of the object files being linked reference symtab. However, suppose you plan to relink the output module and you 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 rts2800.lib for the member that defines symtab and to link in the member.

```
cl2000 −v28 −z −u symtab file1.obj file2.obj rts2800.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 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 7-29. For more information about the default actions of the linker, see section 7.13 on page 7-51.

### 7.4.19 Exhaustively Read and Search Libraries (−x and −priority Options)

There are two ways to exhaustively search for unresolved symbols:

- ❑ Reread libraries if you cannot resolve a symbol reference (−x).
- ❑ Search libraries in the order that they are specified (−priority).

The linker normally reads input files, including archive libraries, only once when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library, the reference is not resolved.

With the −x option, you can force the linker to reread all libraries. The linker rereads libraries until no more references can be resolved. Linking using −x may be slower, so you should use it only as needed. For example, if a.lib contains a reference to a symbol defined in b.lib, and b.lib contains a reference to a symbol defined in a.lib, you can resolve the mutual dependencies by listing one of the libraries twice, as in:

```
cl2000 −v28 −z −la.lib −lb.lib −la.lib
```

or you can force the linker to do it for you:

```
cl2000 −v28 −z −x −la.lib −lb.lib
```

The −priority option provides an alternate search mechanism for libraries. Using −priority causes each unresolved reference to be satisfied by the first library that contains a definition for that symbol. For example:

```
  objfile    references A
  lib1       defines B
  lib2       defines A, B; obj defining A references B

  % cl2000 −v28 −z objfile lib1 lib2
```

Under the existing model, objfile resolves its reference to A in lib2, pulling in a reference to B, which resolves to the B in lib2.

Under −priority, objfile resolves its reference to A in lib2, pulling in a reference to B, but now B is resolved by searching the libraries in order and resolves B to the first definition it finds, namely the one in lib1.

The −priority option is useful for libraries that provide overriding definitions for related sets of functions in other libraries without having to provide a complete version of the whole library.

For example, suppose you want to override versions of malloc and free defined in the rts55.lib without providing a full replacement for rts55.lib. Using −priority and linking your new library before rts55.lib guarantees that all references to malloc and free resolve to the new library.

The −priority option is intended to support linking programs with DSP/BIOS where situations like the one illustrated above occur.

### 7.4.20 Generate XML Link Information File (−−xml_link_info Option)

The linker supports the generation of an XML link information file via the −−xml_link_info *file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker generated map file.

See Appendix C, *XML Link Information File Description*, for specifics on the contents of the generated file.

## 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 7-24). The SECTIONS directive controls how sections are built and allocated (see section 7.8, *The SECTIONS Directive*, on page 7-29.)
- ❑ Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the cl2000 −v28 −z command and follow it with the name of the command file:

**cl2000 −v28 −z** *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:

```
cl2000 −v28 −z link.cmd
```

You can place other parameters on the command line when you use a command file:

```
cl2000 −v28 −z −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:

```
cl2000 −v28 −z 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 7-24. For more information about the SECTIONS directive, see section 7.8, *The SECTIONS Directive*, on page 7-29.

### *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 3.6, *Constants*, on page 3-13 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 7-17). 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:

```
cl2000 −v28 −z 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:

```
cl2000 −v28 −z 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 7-16.) The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
cl2000 −v28 −z −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 7-29.

Section 7.4.9, *Alter the Library Search Algorithm (−l Option, −*`I` *Option, and C_DIR Environment Variable)*, on page 7-10 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 TMS320C28x systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use MEMORY to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

For more information, see section 2.3, *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 TMS320C28x architecture. For more information about the default memory model, see section 7.13, *Default Allocation Algorithm*, on page 7-51.

### 7.7.2 MEMORY Directive Syntax

The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each range has several characteristics:

- Name
- Starting address
- Length
- Optional set of attributes
- Optional fill specification

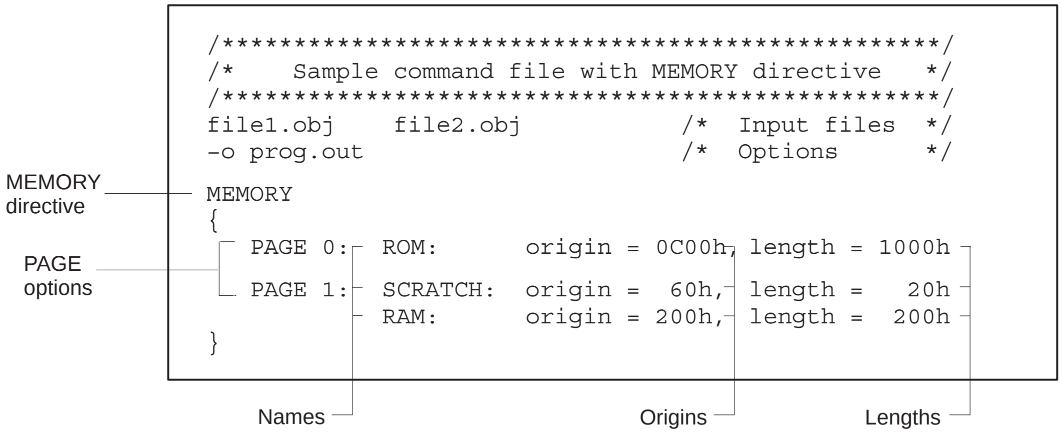
TMS320C28x devices have separate memory spaces (pages) that occupy the same address ranges (overlay). In the default memory map, one space is dedicated to the program area, while a second is dedicated to the data area. (For detailed information about overlaying pages, see section 7.11, *OverlayingPages*, on page 7-46.

In the linker command file, you configure the address spaces separately by using the MEMORY directive's PAGE option. The linker treats each page as a separate memory space. The TMS320C28x supports up to 255 address spaces, but the number of address spaces available depends on the customized configuration of your device (see the *TMS320C28x 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.

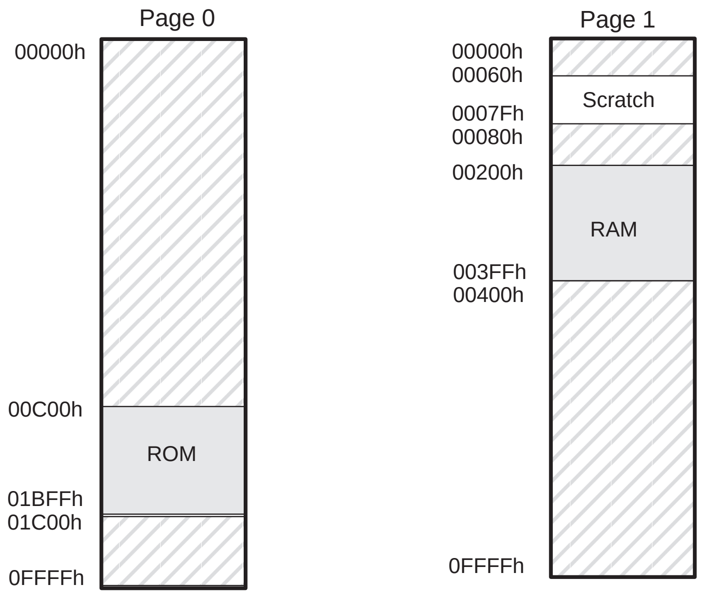
*Example 7−3. The MEMORY Directive*



```
/****************************************************/
/*    Sample command file with MEMORY directive     */
/****************************************************/
file1.obj    file2.obj           /*  Input files  */
-o prog.out                      /*  Options      */

MEMORY
{
   PAGE 0:  ROM:      origin = 0C00h, length = 1000h
   PAGE 1:  SCRATCH:  origin =   60h, length =   20h
            RAM:      origin = 200h,  length =  200h
}
```

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 7-29.

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 7-51 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 7-50.

- ❑ **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 7-61.

Example 7−4 shows a SECTIONS directive in a sample linker command file.

*Example 7−4. The SECTIONS Directive*

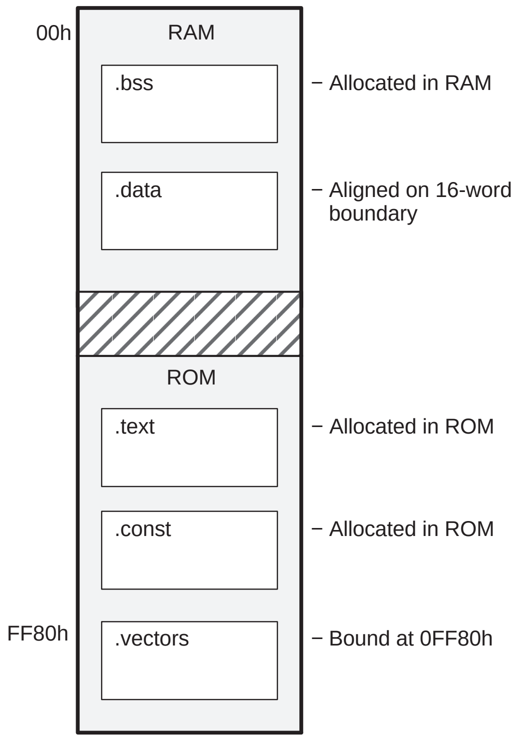
SECTIONS directive

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 7-39.

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to allocate the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run allocation 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 7-36.

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 7-24). 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, the following statement allocates .text so that it falls on a 128-word boundary.:

```
.text: load = align(128)
```

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, the following statement allocates .bss so that the entire section is contained in a single 64K-word page or begins on a page:

```
bss: load = block(0x40)
```

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.

### 7.8.4 Allocating an Archive Member to an Output Section

The linker allows you to allocate one or more members of an archive library into a specific output section. The syntax for such an allocation is:

```
SECTIONS
{
   .output_sec
   {
     [−l] lib_name<obj1 [obj2...objn]> (.sec_name)
    }
}
```

In this syntax, the *lib_name* is the archive library. The −l is optional, since the library search algorithm is always used to search for the archive. For more information on the −l option, see Section 7.4.9, *Alter the Library Search Algorithm*, on page 7-10. Brackets (<>) are used to specify the archive member(s). The brackets may contain one or more object files, separated by a space. The *sec_name* is the archive section to be allocated.

For example:

```
SECTIONS
{
   .boot > BOOT1
   {
     −l rts2800.lib<boot.obj exit.obj strcpy.obj> (.text)
    }
   .rts > BOOT2
   {
     −l rts2800.lib (.text)
   }
   .text > RAM
   {
      * (.text)
   }
}
```

In the specification above, the .text sections of boot.obj, exit.obj, and strcpy.obj from rts2800.lib will be placed in the .boot section.

The remainder of the .text sections from rts2800.lib will be placed in the .rts section.

All other unallocated .text sections will be placed in the .text 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 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 7-43.)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. Everything related to allocation after the keyword *load* affects the load address until the keyword *run* is 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 4-52.

Example 7−6 shows the use of the .label directive.

### Example 7−6. Copying a Section From ROM to RAM

*(a) Assembly language file*

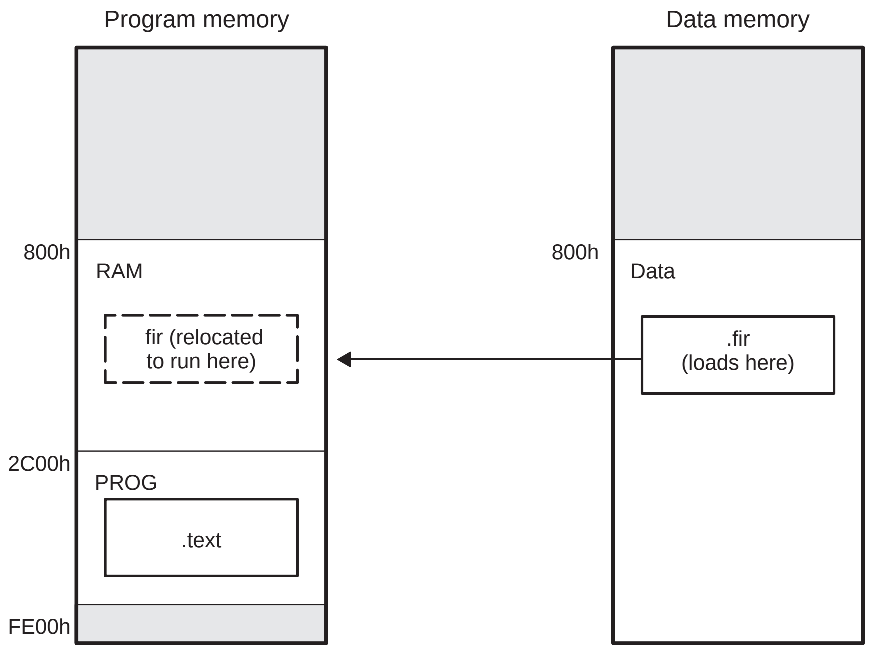
```
;-------------------------------------------------------------
;  define a section to be copied from DATA to PROGRAM
;-------------------------------------------------------------
    .sect  ”.fir”
    .label fir_src        ; load address of section
fir:                      ; run address of section
    <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 illustrates the run-time execution of Example 7−6.

*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*

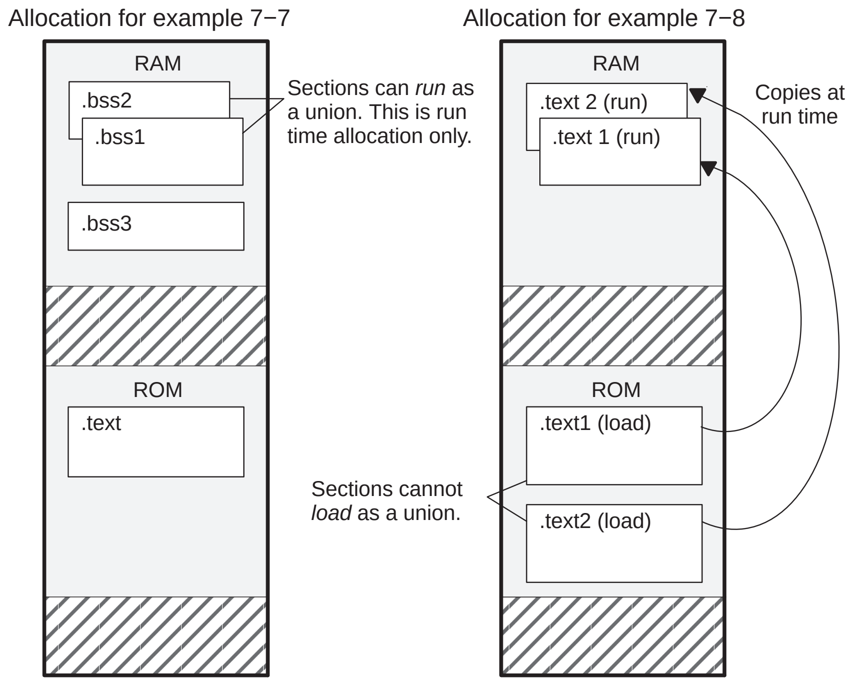
```
SECTIONS
{
   .text: load = ROM
   UNION: run = RAM
   {
      .bss1: { file1.obj(.bss) }
      .bss2: { file2.obj(.bss) }
   }
      .bss3: run = RAM { globals.obj(.bss) }
```

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*


Allocation for example 7−7
Allocation for example 7−8
RAM
.bss2
.bss1
.bss3
ROM
.text
Sections can *run* as a union. This is run time allocation only.
RAM
.text 2 (run)
.text 1 (run)
Copies at run time
ROM
.text1 (load)
.text2 (load)
Sections cannot *load* as a union.

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 7-43) 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*

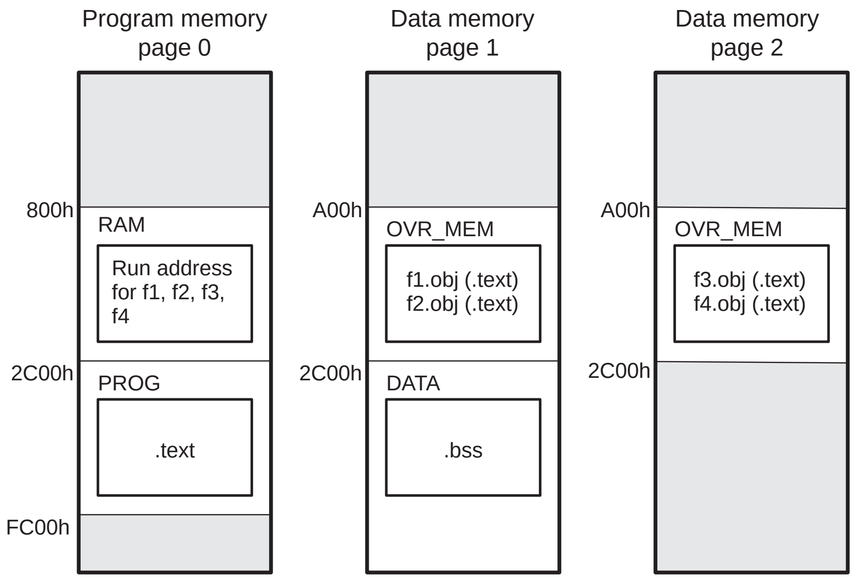
```
SECTIONS
{
   UNION :  run = RAM
   {
      S1 :  load = OVR_MEM PAGE 1
      {
         s1_load = 0A00h;
         s1_start = .;
         f1.obj (.text)
         f2.obj (.text)
         s1_length = . – s1_start;
      }
      S2 :  load = OVR_MEM PAGE 2
      {
         s2_load = 0A00h;
         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 TMS320C28x C/C++ compiler has this attribute under the run-time initialization model.

❑ A NOLOAD section differs from a normal output section in one respect: 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 TMS320C28x Devices*

```
MEMORY
{
   PAGE 0: PROG:  origin = 0x000040  length = 0x3fffc0
   PAGE 1: DATA:  origin = 0x000000  length = 0x010000
   PAGE 1: DATA1: origin = 0x010000  length = 0x3f0000

}
SECTIONS
{
   .text:    PAGE = 0
   .data:    PAGE = 0
   .cinit:   PAGE = 0   /* Used only for C programs */
   .bss:     PAGE = 1
}
```

All .text input sections are concatenated to form a .text output section in the executable output file, and all .data input sections are combined to form a .data output section.

### 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 7-29 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 7-16) 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 7-24, 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 7-29.)

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 4-46), 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 7-61.

### 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 4-46). You must have used the corresponding section in a source module for the symbol to be created. Values are assigned to these symbols as follows:

| | |
|---|---|
| **.text** | is assigned the first address of the .text output section. (It marks the *beginning* of executable code.) |
| **etext** | is assigned the first address following the .text output section. (It marks the *end* of executable code.) |
| **.data** | is assigned the first address of the .data output section. (It marks the *beginning* of initialized data tables.) |
| **edata** | is assigned the first address following the .data output section. (It marks the *end* of initialized data tables.) |
| **.bss** | is assigned the first address of the .bss output section. (It marks the *beginning* of uninitialized data.) |
| **end** | is assigned the first address following the .bss output section. (It marks the *end* of uninitialized data.) |

The following symbols are defined only for C/C++ support when the −c or −cr option is used.

| | |
|---|---|
| **__STACK_SIZE** | is assigned the size of the .stack section. |
| **__SYSMEM_SIZE** | is assigned the size of the .sysmem section. |

### 7.14.5 Assigning Exact Start, End, and Size Values of a Section to a Symbol

The code generation tools currently support the ability to load program code in one area of (slow) memory and run it in another (faster) area. This is done by specifying separate load and run addresses for an output section or group in the linker command file. Then execute a sequence of instructions (the copying code in Example 7−6) that moves the program code from its load area to its run area before it is needed.

There are several responsibilities that a programmer must take on when setting up a system with this feature. One of these responsibilities is to determine the size and run-time address of the program code to be moved. The current mechanisms to do this involve use of the .label directives in the copying code. A simple example is illustrated Example 7−6.

This method of specifying the size and load address of the program code has limitations. While it works fine for an individual input section that is contained entirely within one source file, this method becomes more complicated if the program code is spread over several source files or if the programmer wants to copy an entire output section from load space to run space.

Another problem with this method is that it does not account for the possibility that the section being moved may have an associated far call trampoline section that needs to be moved with it.

### 7.14.6 Why the Dot Operator Does Not Always Work

The dot operator (`.`) is used to define symbols at link-time with a particular address inside of an output section. It is interpreted like a PC. Whatever the current offset within the current section is, that is the value associated with the dot. Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.obj(.text)
        end_of_s2 = .;
}
```

This statement creates three symbols:

- end_of_s1—the end address of .text in s1.obj
- start_of_s2—the start address of .text in s2.obj
- end_of_s2—the end address of .text in s2.obj

Suppose there is padding between s1.obj and s2.obj that is created as a result of alignment. Then start_of_s2 is not really the start address of the .text section in s2.obj, but it is the address before the padding needed to align the .text section in s2.obj. This is due to the linker's interpretation of the dot operator as the current PC. It is also due to the fact that the dot operator is evaluated independently of the input sections around it.

Another potential problem in the above example is that end_of_s2 may not account for any padding that was required at the end of the output section. You cannot reliably use end_of_s2 as the end address of the output section. One way to get around this problem is to create a dummy section immediately after the output section in question. For example:

```
GROUP
{
        outsect:
        {
                start_of_outsect = .;
                ...
        }
        dummy: { size_of_outsect = . - start_of_outsect; }
}
```

### 7.14.7 Address and Dimension Operators

Six new operators have been added to the linker command file syntax:

| | |
|---|---|
| **LOAD_START(***sym***)**<br>**START(***sym***)** | Defines *sym* with the load-time start address of related allocation unit |
| **LOAD_END(***sym***)**<br>**END(***sym***)** | Defines *sym* with the load-time end address of related allocation unit |
| **LOAD_SIZE(***sym***)**<br>**SIZE(***sym***)** | Defines *sym* with the load-time size of related allocation unit |
| **RUN_START(***sym***)** | Defines *sym* with the run-time start address of related allocation unit |
| **RUN_END(***sym***)** | Defines *sym* with the run-time end address of related allocation unit |
| **RUN_SIZE**(*sym***)** | Defines *sym* with the run-time size of related allocation unit |

**Note: Linker Command File Operator Equivalencies**

LOAD_START() and START() are equivalent, as are LOAD_END()/END() and LOAD_SIZE()/SIZE().

The new address and dimension operators can be associated with several different kinds of allocation units, including input items, output sections, GROUPs, and UNIONs. The following sections provide some examples of how the operators can be used in each case.

#### 7.14.7.1 Input Items

Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.obj(.text)
        end_of_s2 = .;
}
```

This can be rewritten using the START and END operators as follows:

```
outsect:
{
        s1.obj(.text) { END(end_of_s1) }
        s2.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

The values of end_of_s1 and end_of_s2 will be the same as if you had used the dot operator in the original example, but start_of_s2 would be defined after any necessary padding that needs to be added between the two .text sections. Remember that the dot operator would cause start_of_s2 to be defined before any necessary padding is inserted between the two input sections.

The syntax for using these operators in association with input sections calls for braces { } to enclose the operator list. The operators in the list are applied to the input item that occurs immediately before the list.

#### 7.14.7.2 Output Section

The START, END, and SIZE operators can also be associated with an output section. Here is an example:

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
        <list of input items>
}
```

In this case, the SIZE operator defines size_of_outsect to incorporate any padding that is required in the output section to conform to any alignment requirements that are imposed.

The syntax for specifying the operators with an output section do not require braces to enclose the operator list. The operator list is simply included as part of the allocation specification for an output section.

#### *7.14.7.3 GROUPs*

Here is another use of the START and SIZE operators in the context of a GROUP specification:

```
GROUP
{
        outsect1: { ... }
        outsect2: { ... }
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

This can be useful if the whole GROUP is to be loaded in one location and run in another. The copying code can use group_start and group_size as parameters for where to copy from and how much is to be copied. This makes the use of .label in the source code unnecessary.

#### *7.14.7.4 UNIONs*

The RUN_SIZE and LOAD_SIZE operators provide a mechanism to distinguish between the size of a UNION's load space and the size of the space where its constituents are going to be copied before they are run. Here is an example:

```
UNION: run = RAM, LOAD_START(union_load_addr),
       LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
        .text1: load = ROM, SIZE(text1_size) { f1.obj(.text) }
        .text2: load = ROM, SIZE(text2_size) { f2.obj(.text) }
}
```

Here union_ld_sz is going to be equal to the sum of the sizes of all output sections placed in the union. The union_run_sz value is equivalent to the largest output section in the union. Both of these symbols incorporate any padding due to blocking or alignment requirements.

## 7.15 Creating and Filling Holes

The 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 4−25) and sections defined with the .usect directive (see page 4-80) 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 7-24.

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 7-53.

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 7-9). 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:

   ```
   cl2000 −v28 −z −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 Linker-Generated Copy Tables

The linker supports extensions to the linker command file syntax that enable the following:

- ❑ Make it easier for you to copy objects from load-space to run-space at boot time
- ❑ Make it easier for you to manage memory overlays at run time
- ❑ Allow you to split GROUPs and output sections that have separate load and run addresses

### 7.16.1 A Current Boot-Loaded Application Development Process

In some embedded applications, there is a need to copy or download code and/or data from one location to another at boot time before the application actually begins its main execution thread. For example, an application may have its code and/or data in FLASH memory and need to copy it into on-chip memory before the application begins execution.

One way you can develop an application like this is to create a copy table in assembly code that contains three elements for each block of code or data that needs to be moved from FLASH into on-chip memory at boot time:

- ❑ The load location (load page id and address)
- ❑ The run location (run page id and address)
- ❑ The size

The process you follow to develop such an application might look like this:

1) Build the application to produce a .map file that contains the load and run addresses of each section that has a separate load and run placement.

2) Edit the copy table (used by the boot loader) to correct the load and run addresses as well as the size of each block of code or data that needs to be moved at boot time.

3) Build the application again, incorporating the updated copy table.

4) Run the application.

This process puts a heavy burden on you to maintain the copy table (by hand, no less). Each time a piece of code or data is added or removed from the application, you must repeat the process in order to keep the contents of the copy table up to date.

### 7.16.2 An Alternative Approach

You can avoid some of this maintenance burden by using the LOAD_START(), RUN_START(), and SIZE() operators that are already part of the linker command file syntax . For example, instead of building the application to generate a .map file, the linker command file can be annotated:

```
SECTIONS
{
   .flashcode: { app_tasks.obj(.text) }
      load = FLASH, run = PMEM,
      LOAD_START(_flash_code_ld_start),
      RUN_START(_flash_code_rn_start),
      SIZE(_flash_code_size)

   ...
}
```

In this example, the LOAD_START(), RUN_START(), and SIZE() operators instruct the linker to create three symbols:

| Symbol | Description |
|---|---|
| _flash_code_ld_start | load address of .flashcode section |
| _flash_code_rn_start | run address of .flashcode section |
| _flash_code_size | size of .flashcode section |

These symbols can then be referenced from the copy table. The actual data in the copy table will be updated automatically each time the application is linked. This approach removes step 1 of the process described in section 7.16.1.

While maintenance of the copy table is reduced markedly, you must still carry the burden of keeping the copy table contents in sync with the symbols that are defined in the linker command file. Ideally, the linker would generate the boot copy table automatically. This would avoid having to build the application twice *and* free you from having to explicitly manage the contents of the boot copy table.

For more information on the LOAD_START(), RUN_START(), and SIZE() operators, see section 7.14.7, *Address and Dimension Operators*, on page 7-58.

### 7.16.3 Overlay Management Example

Consider an application which contains a memory overlay that must be managed at run time. The memory overlay is defined using a UNION in the linker command file as illustrated in Example 7−13:

*Example 7−13. Using a UNION for Memory Overlay*

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1: { task1.obj(.text) }
         .task2: { task2.obj(.text) }

      } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)

      GROUP
      {
         .task3: { task3.obj(.text) }
         .task4: { task4.obj(.text) }

      } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)

   } run = RAM, RUN_START(_task_run_start)

 ...
}
```

The application must manage the contents of the memory overlay at run time. That is, whenever any services from .task1 or .task2 are needed, the application must first ensure that .task1 and .task2 are resident in the memory overlay. Similarly for .task3 and .task4.

To effect a copy of .task1 and .task2 from ROM to RAM at run time, the application must first gain access to the load address of the tasks (_task12_load_start), the run address (_task_run_start), and the size (_task12_size). Then this information is used to perform the actual code copy.

### 7.16.4 Generating Copy Tables Automatically with the Linker

The linker supports extensions to the linker command file syntax that enable you to do the following:

- ❑ Identify any object components that may need to be copied from load space to run space at some point during the run of an application
- ❑ Instruct the linker to automatically generate a copy table that contains (at least) the load address, run address, and size of the component that needs to be copied
- ❑ Instruct the linker to generate a symbol specified by you that provides the address of a linker-generated copy table. For instance, Example 7−13 can be written as shown in Example 7−14:

*Example 7−14. Produce Address for Linker Generated Copy Table*

```
SECTIONS
{
   ...

   UNION
   {
      GROUP
      {
         .task1: { task1.obj(.text) }
         .task2: { task2.obj(.text) }

      } load = ROM, table(_task12_copy_table)

      GROUP
      {
         .task3: { task3.obj(.text) }
         .task4: { task4.obj(.text) }

      } load = ROM, table(_task34_copy_table)

   } run = RAM

   ...
}
```

Using the SECTIONS directive from Example 7−14 in the linker command file, the linker generates two copy tables named: _task12_copy_table and _task34_copy_table. Each copy table provides the load page id, run page id, load address, run address, and size of the GROUP that is associated with the copy table. This information is accessible from application source code using the linker-generated symbols, _task12_copy_table and _task34_copy_table, which provide the addresses of the two copy tables, respectively.

Using this method, you do not have to worry about the creation or maintenance of a copy table. You can reference the address of any copy table generated by the linker in C/C++ or assembly source code, passing that value to a general purpose copy routine which will process the copy table and effect the actual copy.

### 7.16.5 The table() Operator

You can use the table() operator to instruct the linker to produce a copy table. A table() operator can be applied to an output section, a GROUP, or a UNION member. The copy table generated for a particular table() specification can be accessed through a symbol specified by you that is provided as an argument to the table() operator. The linker creates a symbol with this name and assigns it the address of the copy table as the value of the symbol. The copy table can then be accessed from the application using the linker-generated symbol.

Each table() specification you apply to members of a given UNION must contain a unique name. If a table() operator is applied to a GROUP, then none of that GROUP's members may be marked with a table() specification. The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table() specification. The linker does not generate a copy table for erroneous table() operator specifications.

### 7.16.6 Boot-Time Copy Tables

The linker supports a special copy table name, BINIT (or binit), that you can use to create a boot-time copy table. For example, the linker command file for the boot-loaded application described in section 7.16.2 can be rewritten as follows:

```
SECTIONS
{
   .flashcode: { app_tasks.obj(.text) }
   load = FLASH, run = PMEM,
      table(BINIT)
   ...
}
```

For this example, the linker creates a copy table that can be accessed through a special linker-generated symbol, ___binit__, which contains the list of all object components that need to be copied from their load location to their run location at boot-time. If a linker command file does not contain any uses of table(BINIT), then the ___binit__ symbol is given a value of −1 to indicate that a boot-time copy table does not exist for a particular application.

You can apply the table(BINIT) specification to an output section, GROUP, or UNION member. If used in the context of a UNION, only one member of the UNION can be designated with table(BINIT). If applied to a GROUP, then none of that GROUP's members may be marked with table(BINIT).The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table(BINIT) specification.

### 7.16.7 Using the table() Operator to Manage Object Components

If you have several pieces of code that need to be managed together, then you can apply the same table() operator to several different object components. In addition, if you want to manage a particular object component in multiple ways, you can apply more than one table() operator to it. Consider the linker command file excerpt in Example 7−15:

*Example 7−15. Linker Command File to Manage Object Components*

```
SECTIONS
{
   UNION
   {
      .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second: { a2.obj(.text), b2.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra: load = EMEM, run = PMEM, table(BINIT)

   ...
}
```

In this example, the output sections .first and .extra are copied from external memory (EMEM) into program memory (PMEM) at boot time while processing the BINIT copy table. After the application has started executing its main thread, it can then manage the contents of the overlay using the two overlay copy tables named: _first_ctbl and _second_ctbl.

### 7.16.8 Copy Table Contents

In order to use a copy table that is generated by the linker, you must be aware of the contents of the copy table. This information is included in a new run-time-support library header file, cpy_tbl.h, which contains a C source representation of the copy table data structure that is automatically generated by the linker.

Example 7−16 shows the TMS320C28x copy table header file.

*Example 7−16. TMS320C28x cpy_tbl.h File*

```
/*****************************************************************************/
/* cpy_tbl.h                                                                 */
/*                                                                           */
/* Copyright (c) 2003 Texas Instruments Incorporated                         */
/*                                                                           */
/* Specification of copy table data structures which can be automatically   */
/* generated by the linker (using the table() operator in the LCF).          */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/* Copy Record Data Structure                                                */
/*****************************************************************************/
typedef struct copy_record
{
   unsigned int          src_pgid;
   unsigned int          dst_pgid;
   unsigned long         src_addr;
   unsigned long         dst_addr;
   unsigned long         size;
} COPY_RECORD;

/*****************************************************************************/
/* Copy Table Data Structure                                                 */
/*****************************************************************************/
typedef struct copy_table
{
   unsigned int          rec_size;
   unsigned int          num_recs;
   COPY_RECORD           recs[1];
} COPY_TABLE;

/*****************************************************************************/
/* Prototype for general purpose copy routine.                               */
/*****************************************************************************/
extern void copy_in(COPY_TABLE *tp);

/*****************************************************************************/
/* Prototypes for utilities used by copy_in() to move code/data between     */
/* program and data memory (see cpy_utils.asm for source).                   */
/*****************************************************************************/
extern void ddcopy(unsigned long src, unsigned long dst);
extern void dpcopy(unsigned long src, unsigned long dst);
extern void pdcopy(unsigned long src, unsigned long dst);
extern void ppcopy(unsigned long src, unsigned long dst);
```

For each object component that is marked for a copy, the linker creates a COPY_RECORD object for it. Each COPY_RECORD contains at least the following information for the object component:

- The load page id
- The run page id
- The load address
- The run address
- The size

The linker collects all COPY_RECORDs that are associated with the same copy table into a COPY_TABLE object. The COPY_TABLE object contains the size of a given COPY_RECORD, the number of COPY_RECORDs in the table, and the array of COPY_RECORDs in the table. For instance, in the BINIT example in section 7.16.6, the .first and .extra output sections will each have their own COPY_RECORD entries in the BINIT copy table. The BINIT copy table will then look like this:

```
COPY_TABLE __binit__ = { 12, 2,
                           { <load page id of .first>,
                             <run page id of .first>,
                             <load address of .first>,
                             <run address of .first>,
                             <size of .first> },
                           { <load page id of .extra>,
                             <run page id of .extra>,
                             <load address of .extra>,
                             <run address of .extra>,
                             <size of .extra> } };
```

### 7.16.9 General Purpose Copy Routine

The cpy_tbl.h file in Example 7−16 also contains a prototype for a general-purpose copy routine, copy_in(), which is provided as part of the run-time-support library. The copy_in() routine takes a single argument: the address of a linker-generated copy table. The routine then processes the copy table data object and performs the copy of each object component specified in the copy table.

The copy_in() function definition is provided in the cpy_tbl.c run-time-support source file shown in Example 7−17.

*Example 7−17. Run-Time-Support cpy_tbl.c File*

```
/****************************************************************************/
/* cpy_tbl.c                                                                */
/*                                                                          */
/* Copyright (c) 2003 Texas Instruments Incorporated                        */
/*                                                                          */
/* General purpose copy routine.  Given the address of a linker-generated   */
/* COPY_TABLE data structure, effect the copy of all object components      */
/* that are designated for copy via the corresponding LCF table() operator. */
/*                                                                          */
/****************************************************************************/
#include <cpy_tbl.h>
#include <string.h>

/****************************************************************************/
/* COPY_IN()                                                                */
/****************************************************************************/
void copy_in(COPY_TABLE *tp)
{
   unsigned int i;
   for (i = 0; i < tp->num_recs; i++)
   {
      COPY_RECORD *crp = &tp->recs[i];
      unsigned int cpy_type = 0;
      unsigned int j;

      if (crp->src_pgid) cpy_type += 2;
      if (crp->dst_pgid) cpy_type += 1;

      for (j = 0; j < crp->size; j++)
      {
         switch (cpy_type)
         {
            case 3: ddcopy(crp->src_addr + j, crp->dst_addr + j); break;
            case 2: dpcopy(crp->src_addr + j, crp->dst_addr + j); break;
            case 1: pdcopy(crp->src_addr + j, crp->dst_addr + j); break;
            case 0: ppcopy(crp->src_addr + j, crp->dst_addr + j); break;
         }
      }
   }
}
```

The load (or source) page id and the run (or destination) page id are used to choose which low-level copy routine is called to move a word of data from the load location to the run location. A page id of 0 indicates that the specified address is in program memory, and a page id of 1 indicates that the address is in data memory. The hardware provides special instructions, PREAD and PWRITE, to move code/data into and out of program memory.

The source for the low-level copy routines is included in another run-time-support source file called cpy_utils.asm as shown in Example 7−18.

*Example 7−18. The cpy_utils.asm File*

```
;*********************************************************************
; CPY_UTILS.ASM
;
; Copy utility functions to perform 4 different types of copy,
; data -> data, data -> program, program -> data, and
; program -> program.
;
; source address always arrives in the accumulator 'ACC'
; destination address arrives on the stack '*-SP[2]'
;
; We will use XAR6 and XAR7 as temporary registers in
; these routines.
;
;*********************************************************************

        .if     .TMS320C2800
        .asg    LRETR, RETURN
        .asg    MOVL,  MOVE_L
        .else
        .asg    RET,   RETURN
        .asg    MOV,   MOVE_L
        .endif

        .sect   ".text"
        .global _ddcopy
;*********************************************************************
; Data Memory -> Data Memory
;
; Use the accumulator to hold the data being moved between the
; load and the store.
;*********************************************************************
_ddcopy:
        ; Load word at Source Address into Accumulator
        MOVL    XAR6,ACC
        MOV     AL,*+XAR6[0]

        ; Store word in Accumulator at Dest Address
        MOVE_L  XAR6,*-SP[4]
        MOV     *+XAR6[0],AL

        RETURN

        .sect   ".text"
        .global _dpcopy
;*********************************************************************
; Data Memory -> Program Memory
;
; Transfer the dest address into XAR7 so that we can use a PWRITE
; instruction to store the data into program memory.
;*********************************************************************
_dpcopy:
        ; Load Dest address into XAR7
        MOVE_L  XAR7,*-SP[4]
```

*Example 7−18. The cpy_utils.asm File (Continued)*

```
        ; Write data to Program Memory
        MOVL    XAR6,ACC
        PWRITE  *XAR7,*XAR6
        RETURN

        .sect   ".text"
        .global _pdcopy
;*********************************************************************
; Program Memory -> Data Memory
;
; Put source address into XAR7 so we can use a PREAD instruction to
; get the data from program memory.
;*********************************************************************
_pdcopy:
        ; Load Source address into XAR7
        MOVL    XAR7,ACC

        ; Read from Program Memory into Dest Address (data memory)
        MOVE_L  XAR6,*-SP[4]
        PREAD   *XAR6,*XAR7

        RETURN

        .sect   ".text"
        .global _ppcopy
;*********************************************************************
; Program Memory -> Program Memory
;
; First read the data from program memory into the accumulator.
; Then store it into the destination location (also in program
; memory).  We use XAR7 to hold first the source address and then
; the dest address so that we can make use of the PREAD and
; PWRITE instructions for reading/writing data from/to program
; memory.
;*********************************************************************
_ppcopy:
        ; Load Source Address into XAR7
        MOVL    XAR7,ACC

        ; Read from Program Memory into Accumulator
        PREAD   @AL,*XAR7

        ; Load Dest address into XAR7
        MOVE_L  XAR7,*-SP[4]

        ; Write Acummulator contents to Program Memory
        PWRITE  *XAR7,@AL

        RETURN
```

### 7.16.10 Linker Generated Copy Table Sections and Symbols

The linker creates and allocates a separate input section for each copy table that it generates. Each copy table symbol is defined with the address value of the input section that contains the corresponding copy table.

The linker generates a unique name for each overlay copy table input section. For example, table(_first_ctbl) would place the copy table for the .first section into an input section called .ovly:_first_ctbl. The linker creates a single input section, .binit, to contain the entire boot-time copy table.

Example 7−19 illustrates how you can control the placement of the linker-generated copy table sections using the input section names in the linker command file.

*Example 7−19. Controlling the Placement of the Linker-Generated Copy Table Sections*

```
SECTIONS
{
   UNION
   {
      .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
             load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

      .second: { a2.obj(.text), b2.obj(.text) }
             load = EMEM, run = PMEM, table(_second_ctbl)
   }

   .extra: load = EMEM, run = PMEM, table(BINIT)

   ...

   .ovly: { } > BMEM
   .binit: { } > BMEM
}
```

For the linker command file in Example 7−19, the boot-time copy table is generated into a .binit input section, which is collected into the .binit output section, which is mapped to an address in the BMEM memory area. The _first_ctbl is generated into the .ovly:_first_ctbl input section and the _second_ctbl is generated into the .ovly:_second_ctbl input section. Since the base names of these input sections match the name of the .ovly output section, the input sections are collected into the .ovly output section, which is then mapped to an address in the BMEM memory area.

If you do not provide explicit placement instructions for the linker-generated copy table sections, they are allocated according to the linker's default placement algorithm.

The linker does not allow other types of input sections to be combined with a copy table input section in the same output section. The linker does not allow a copy table section that was created from a partial link session to be used as input to a succeeding link session.

### 7.16.11 Splitting Object Components and Overlay Management

In previous versions of the linker, splitting sections that have separate load and run placement instructions was not permitted. This restriction was because there was no effective mechanism for you, the developer, to gain access to the load address or run address of each one of the pieces of the split object component. Therefore, there was no effective way to write a copy routine that could move the split section from its load location to its run location.

However, the linker can access both the load location and run location of every piece of a split object component. Using the table( ) operator, you can tell the linker to generate this information into a copy table. The linker gives each piece of the split object component a COPY_RECORD entry in the copy table object.

For example, consider an application which has 7 tasks. Tasks 1 through 3 are overlaid with tasks 4 through 7 (using a UNION directive). The load placement of all of the tasks is split among 4 different memory areas (LMEM1, LMEM2, LMEM3, and LMEM4). The overlay is defined as part of memory area PMEM. You must move each set of tasks into the overlay at run time before any services from the set are used.

You can use table( ) operators in combination with splitting operators, >>, to create copy tables that have all the information needed to move either group of tasks into the memory overlay as shown in Example 7−20. Example 7−21 illustrates a possible driver for such an application.

*Example 7−20. Creating a Copy Table to Access a Split Object Component*

```
SECTIONS
{
   UNION
   {
      .task1to3: { *(.task1), *(.task2), *(.task3) }
              load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)

      GROUP
      {
         .task4: { *(.task4) }
         .task5: { *(.task5) }
         .task6: { *(.task6) }
         .task7: { *(.task7) }

      } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)

   } run = PMEM

   ...

   .ovly: > LMEM4
}
```

*Example 7−21. Split Object Component Driver*

```
#include <cpy_tbl.h>

extern COPY_TABLE task13_ctbl;
extern COPY_TABLE task47_ctbl;

extern void task1(void);
...
extern void task7(void);

main()
{
   ...
   copy_in(&task13_ctbl);
   task1();
   task2();
   task3();
   ...

   copy_in(&task47_ctbl);
   task4();
   task5();
   task6();
   task7();
   ...
}
```

The contents of the .task1to3 section are split in the section's load space and contiguous in its run space. The linker-generated copy table, _task13_ctbl, contains a separate COPY_RECORD for each piece of the split section .task1to3. When the address of _task13_ctbl is passed to copy_in(), each piece of .task1to3 is copied from its load location into the run location.

The contents of the GROUP containing tasks 4 through 7 are also split in load space. The linker performs the GROUP split by applying the split operator to each member of the GROUP in order. The copy table for the GROUP then contains a COPY_RECORD entry for every piece of every member of the GROUP. These pieces are copied into the memory overlay when the _task47_ctbl is processed by copy_in().

The split operator can be applied to an output section, GROUP, or the load placement of a UNION or UNION member. The linker does not permit a split operator to be applied to the run placement of either a UNION or of a UNION member. The linker detects such violations, emits a warning, and ignores the offending split operator usage.

## 7.17 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as *partial linking* or *incremental linking*. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program.

Follow these guidelines for producing a file that you will relink:

- ❑ The intermediate files produced by the 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 7-6.)
- ❑ 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 7-15.)
- ❑ 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 7-10).
- ❑ 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 7-8.)
- ❑ 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.

```
cl2000 −v28 −z −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.

```
cl2000 −v28 −z −r −o tempout2 file2.com
```

file2.com contains:

```
SECTIONS
{
   ss2:   {
          g1.obj
          g2.obj
           .
           .
           .
          gn.obj
          }
}
```

**Step 3:** Link tempout1.out and tempout2.out.

```
cl2000 −v28 −z −m final.map −o final.out tempout1.out tempout2.out
```

## 7.18 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:

```
cl2000 −v28 −z −c −o prog.out prog1.obj prog2.obj ...
rts2800.lib
```

The −c option tells the linker to use special conventions that are defined by the C environment. The archive library rts2800.lib contains C run-time-support functions.

For more information about C, including the run-time environment and run-time-support functions, see the *TMS320C28x Optimizing C/C++ Compiler User's Guide.*

### 7.18.1 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, rts2800.lib, contains boot.obj. You can:

❑ Include rts2800.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

### 7.18.2 Object Libraries and Run-Time Support

The *TMS320C28x Optimizing C/C++ Compiler User's Guide* describes additional run-time-support functions that are included in rts.src. If your program uses any of these functions, you must link *rts2800.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.

### 7.18.3 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 7-10 and section 7.4.16, *Define Stack Size (−stack* Size *Option)*, on page 7-15 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.

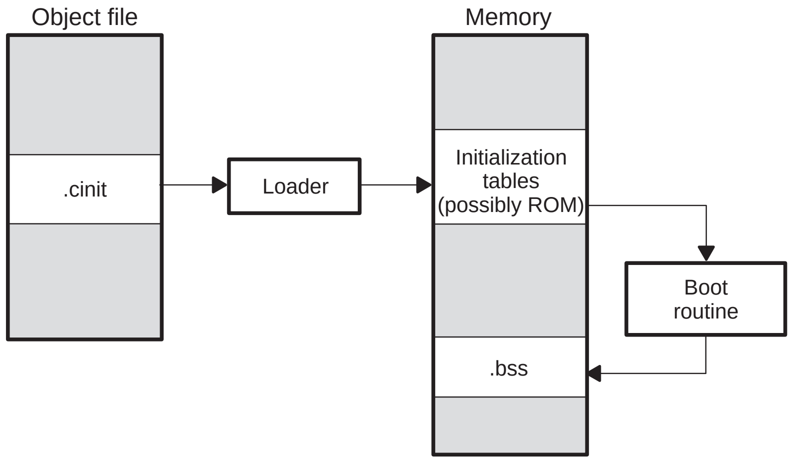
### 7.18.4 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the 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*



### 7.18.5 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:

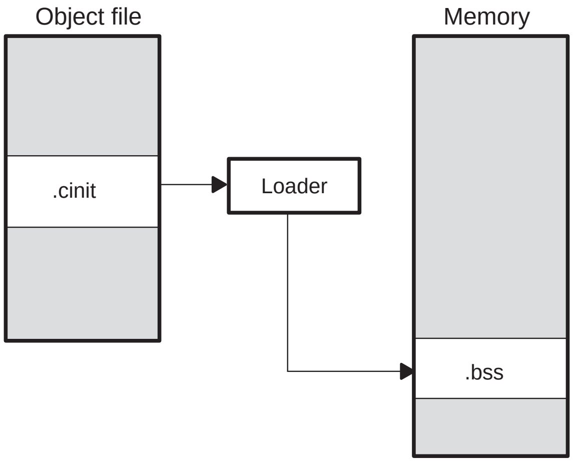
❑ Detect the presence of the .cinit section in the object file

❑ 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*



### 7.18.6 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 rts2800.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 TMS320C28x Code Composer Studio as a loader.

## 7.19 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 7−22 shows the linker command file for this example. Example 7−23 shows the map file.

*Example 7−22. 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:

**cl2000 −v28 −z** `demo.cmd`

This creates the map file shown in Example 7−23 and an output file called demo.out that can be run on a TMS320C28x device.

*Example 7−23. 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 7−23. 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 TMS320C28x™ absolute lister is a debugging tool that accepts linked object files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code. Manually, this could be a tedious process requiring many operations; however, the absolute lister utility performs these operations automatically.

**Topic** **Page**

**8.1 Producing an Absolute Listing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-2**

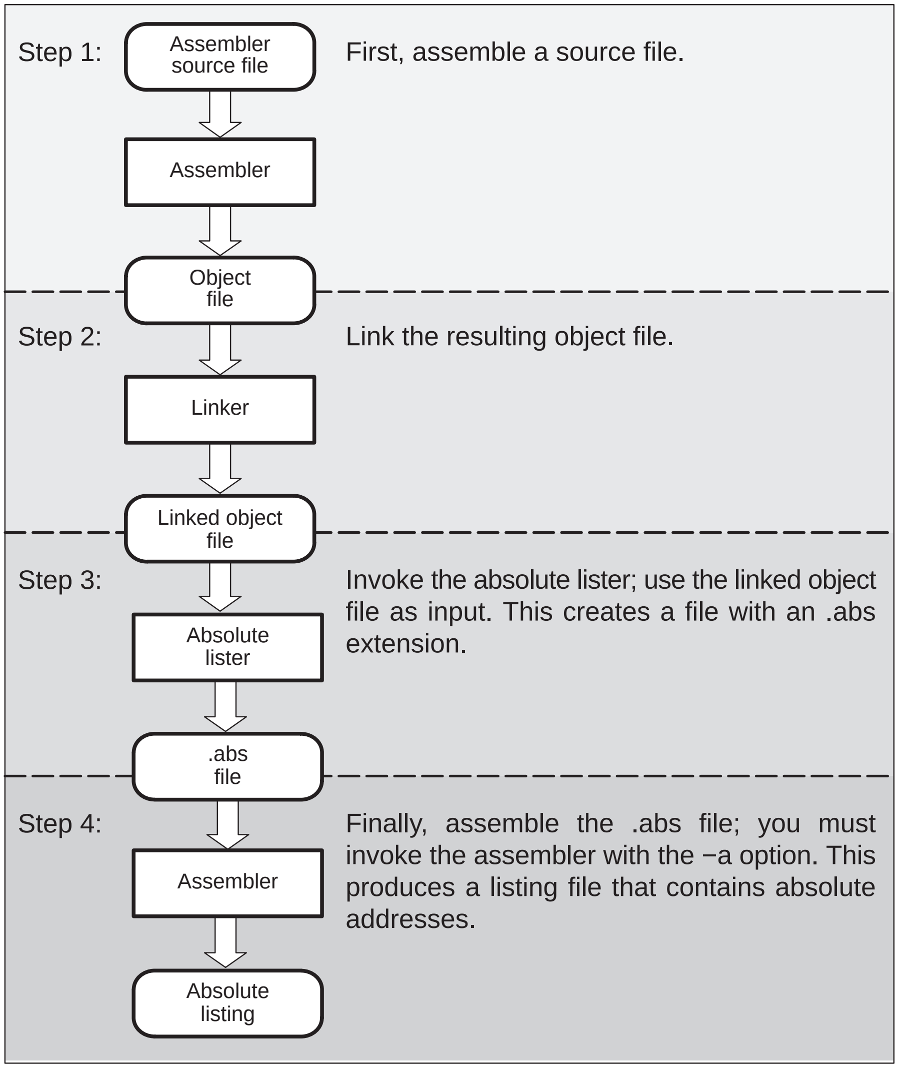
**8.2 Invoking the Absolute Lister . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-3**

**8.3 Absolute Lister Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-5**

## 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*



First, assemble a source file.

Link the resulting object file.

Invoke the absolute lister; use the linked object file as input. This creates a file with an .abs extension.

Finally, assemble the .abs file; you must invoke the assembler with the −a option. This produces a listing file that contains absolute addresses.

## 8.2 Invoking the Absolute Lister

The syntax for invoking the absolute lister is:

**abs2000** [−*options*] *input file*

| | |
|---|---|
| **abs2000** | is the command that invokes the absolute lister. |
| *options* | identifies the absolute lister options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen (−). The absolute lister options are as follows: |

**−e** enables you to change the default naming conventions for filename extensions on assembly files, C source files, and C header files. The three options are listed below.

| | |
|---|---|
| ❑ −ea [.]*asmext* | for assembly files (default is .asm) |
| ❑ −ec [.]*cext* | for C source files (default is .c) |
| ❑ −eh [.]*hext* | for C header files (default is .h) |
| ❑ −ep[.]cppext | for C++ source files (default is .cpp, .cc and .cxx) |

The . in the extensions and the space between the option and the extension are optional.

**−fs** Specifies a directory for the output files. For example, to place the .abs file generated by the absolute lister in the C:\ABSDIR use the command abs2000 −fs C:\ABSDIR filename.out. If the −fs option is not specified, the absolute lister generates the .abs files in the current directory.

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

*input file* names the linked object file. If you do not supply an extension, the absolute lister assumes that the input file has the default extension .out. If you do not supply an input filename when you invoke the absolute lister, the absolute lister prompts you for one.

The absolute lister produces an output file for each file that was linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

Assemble these files with the −a assembler option as follows to create the absolute listing:

```
cl2000 -v28 -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:

```
cl2000 –v28 module1
cl2000 –v28 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 TMS320C28x 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:

**`cl2000 –v28 –z bttest.cmd`**

This command creates an executable object file called bttest.out; use this new file as input for the absolute lister.

**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    ”.bss”,00008000h
          .list
          .text
          .copy       ”module1.asm”
```

**module2.abs:**

```
          .nolist
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”,00008066h
          .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):

```
cl2000 −v28  −a  module1.abs
cl2000 −v28  −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
```

*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
```

Chapter 9

# Cross-Reference Lister Description

The TMS320C28x™ cross-reference lister is a debugging tool. This utility accepts linked object files as input and produces a cross-reference listing as output. This listing shows symbols, their definitions, and their references in the linked source files.

**Topic** **Page**

**9.1 Producing a Cross-Reference Listing . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-2**

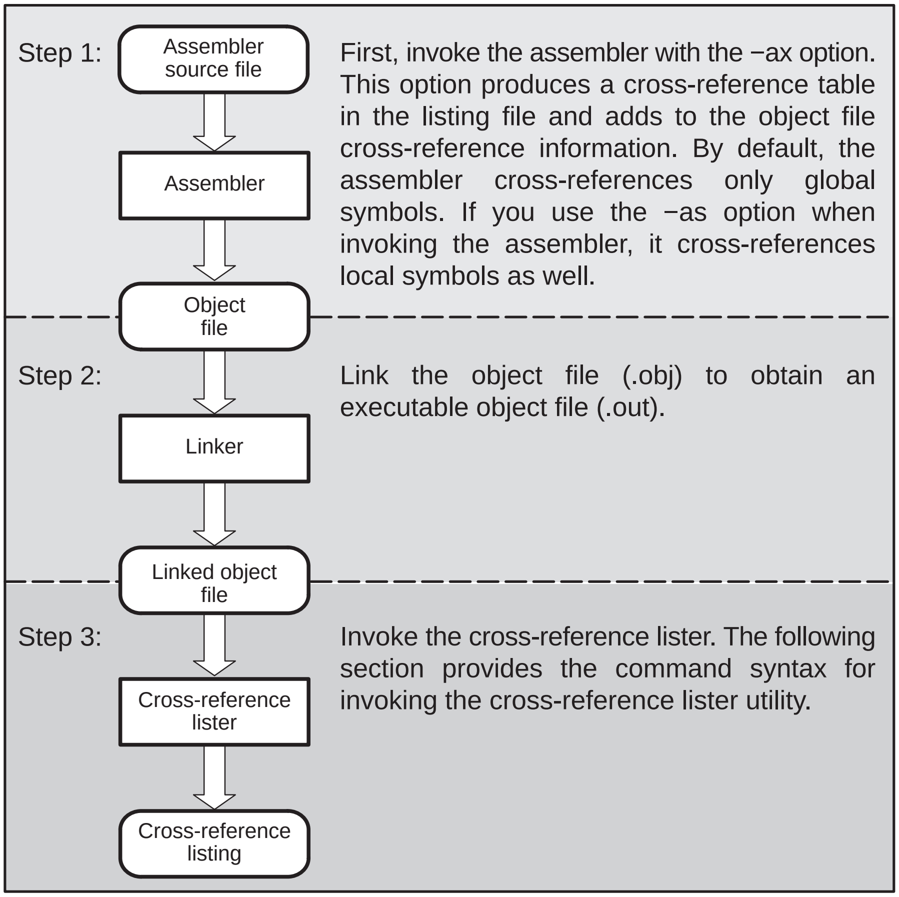
**9.2 Invoking the Cross-Reference Lister . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-3**

**9.3 Cross-Reference Listing Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-4**

## 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*



First, invoke the assembler with the −ax option. This option produces a cross-reference table in the listing file and adds to the object file cross-reference information. By default, the assembler cross-references only global symbols. If you use the −as option when invoking the assembler, it cross-references local symbols as well.

Link the object file (.obj) to obtain an executable object file (.out).

Invoke the cross-reference lister. The following section provides the command syntax for invoking the cross-reference lister utility.

## 9.2 Invoking the Cross-Reference Lister

To use the cross-reference utility, the file must be assembled with the correct options and then linked into an executable file. Assemble the assembly language files with the −ax option. This option 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 −as 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:<br>**−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.<br>**−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

# Object File Utilities Descriptions

This chapter describes how to invoke the following miscellaneous utilities:

❑ The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both human readable and XML formats.

❑ The **name utility** prints a list of names defined and referenced in a COFF object or an executable file.

❑ The **strip utility** removes symbol table and debugging information from object and executable files.

**Topic** **Page**

**10.1 Invoking the Object File Display Utility . . . . . . . . . . . . . . . . . . . . . . . . . 10-2**
**10.2 XML Tag Index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-3**
**10.3 Example XML Consumer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-9**
**10.4 Invoking the Name Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-16**
**10.5 Invoking the Strip Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-17**

## 10.1 Invoking the Object File Display Utility

The object file display utility, *ofd2000*, is used to print the contents of object files (.obj), executable files (.out), and/or archive libraries (.lib) in both human readable and XML formats.

To invoke the object file display utility, enter the following:

**ofd2000** [−*options*] *input filename* [*input filename*]

| | |
|---|---|
| **ofd2000** | is the command that invokes the object file display utility. |
| *input filenames* | names the assembly language source file. The file name must contain a .asm extension. |
| *options* | identify the object file display utility options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen (−). The object file display utility options are as follows: |

| | |
|---|---|
| **−g** | appends DWARF debug information to program output. |
| **−o***filename* | sends program output to *filename* rather than to the screen. |
| **−x** | displays output in XML format. |

If the object file display utility is invoked without any options, it displays information about the contents of the input files on the console screen.

## 10.2 XML Tag Index

Table 10−1 describes the XML tags that are generated by the object file display utility.

*Table 10−1. XML Tag Index*

| Tag Name | Context | Description |
|---|---|---|
| <addr> | <line_entry> | PC address |
| | <row> | PC address |
| | <value> | Machine address |
| <addr_class> | <value> | Address class |
| <addr_size> | <compile_unit> | Size of one machine address (octets) |
| | <section> | Size of one machine address (octets) |
| <alignment> | <section> | Alignment factor |
| <archive> | <ofd> | Archive file (.lib) |
| <attribute> | <die> | Attribute of a DWARF DIE |
| <aux_count> | <symbol> | Number of auxiliary entries for this symbol |
| <banner> | <ofd> | Tool name and version information |
| <block> | <section> | True if alignment is used as blocking factor |
| | <value> | Data block |
| <bss> | <section> | True if this section contains uninitialized data |
| <bss_size> | <optional_file_header> | Size of uninitialized data |
| <byte_swapped> | <file_header> | Endianness of build host is opposite of current host |
| <clink> | <section> | True if this section is conditionally linked |
| <column> | <line_entry> | Source column number |
| <compile_unit> | <section> | Compile unit |
| <const> | <value> | Constant |
| <copy> | <section> | True if this section is a copy section |
| <copyright> | <ofd> | Copyright notice |
| <cpu_flags> | <file_header> | CPU ags |
| <data> | <section> | True if this section contains initialized data |

*Table 10−1. XML Tag Index*

| Tag Name | Context | Description |
|---|---|---|
| <data_size> | <optional_file_header> | Size of initialized data |
| <data_start> | <optional_file_header> | Beginning address of initialized data |
| <destination> | <register> | Destination register |
| <die> | <compile_unit> | DWARF debugging information entry (DIE) |
| <dim_bound> | <dimension> | Dimension upper-bound |
| <dim_num> | <dimension> | Dimension number |
| <dimension> | <symbol> | Array dimension |
| <disp> | <reloc_entry> | Extra address encoding information |
| <dummy> | <section> | True if this section is a dummy section |
| <dwarf> | <ti_coff> | DWARF information |
| <endian> | <file_header> | Endianness of target machine |
| <entry_point> | <optional_file_header> | Entry point of executable program |
| <exec> | <file_header> | True if this file is executable |
| <fde> | <section> | A DWARF frame description entry (FDE) |
| <field_size> | <reloc_entry> | Size of the field to relocate |
| <file_header> | <ti_coff> | COFF file header |
| <file_length> | <file_header> | Size of this file |
| <file_name> | <line_entry> | Name of source file |
|  | <symbol> | Name of source file |
| <file_offsets> | <section> | File offsets associated with this section |
| <flag> | <value> | Flag |
| <form> | <attribute> | Attribute form |
| <frame_size> | <symbol> | Size of function frame |
| <function> | <line_numbers> | Line number entries for one function |
| <icode> | <section> | True if this section has I-Code associated with it |
| <index> | <symbol> | Index of this symbol in the symbol table |

*Table 10−1. XML Tag Index*

| Tag Name | Context | Description |
|---|---|---|
| <indirect register> | <memory> | Indirect register used for calculating destination address |
| <initial location> | <fde> | Start of function referred to by the FDE |
| <internal> | <reloc_entry> | True if this relocation is internal |
| <kind> | <symbol> | Kind of symbol (defined, undefined, absolute, symbolic debug) |
| <length> | <symbol> | Length of section |
| <line> | <line_entry> | Source line number |
|  | <symbol> | First source line associated with this symbol |
| <line_count> | <section> | Number of line number entries |
|  | <symbol> | Number of line number entries |
| <line_entry> | <compile_unit> | Line number entry |
|  | <line_numbers> | Line number entry |
| <line_numbers> | <section> | Line number entries |
| <line_ptr> | <file_offsets> | File offset of line number entries |
|  | <symbol> | File offset of line number entries |
| <lnno_strip> | <file_header> | True if line numbers were stripped from this file |
| <localsym_strip> | <file_header> | True if local symbols were stripped from this file |
| <magic> | <optional_file_header> | Optional file header magic number (0x0108) |
| <math_relative> | <reloc_entry> | True if this relocation is math relative |
| <memory> | <row> | SOE register is saved to memory |
| <name> | <fde> | Name of function referred to by the FDE |
|  | <function> | Name of the current function |
|  | <ofd> | Name of an object or archive file |
|  | <section> | Name of this section |
|  | <symbol> | Name of this symbol |
| <next_symbol> | <symbol> | Index of next symbol after mutlisymbol entity |

*Table 10−1. XML Tag Index*

| Tag Name | Context | Description |
|---|---|---|
| <noload> | <section> | True if this section is a no-load section |
| <object_file> | <ofd> | Object file (.obj, .out) |
| <ofd> | | Object file display (OFD) document |
| <offset> | <memory> | Offset of destination address from indirect register |
| | <reloc_entry> | Offset of the field from relocatable address |
| <optional_file_header> | <ti_coff> | Optional file header |
| <padded> | <section> | True if this section has been padded (C55x only) |
| <page> | <section> | Memory page |
| <pass> | <section> | True if this section is passed through unchanged |
| <physical_addr> | <section> | Physical (run) address of section |
| <raw_data_ptr> | <file_offsets> | File offset of raw data |
| <raw_data_size> | <section> | Size of raw data (octets) |
| <ref> | <value> | Reference |
| <register> | <row> | SOE register is saved to register |
| <register_mask> | <symbol> | Mask of saved SOE registers |
| <regular> | <section> | True if this section is a regular section |
| <reloc_count> | <section> | Number of relocation entries |
| | <symbol> | Number of relocation entries |
| <reloc_entry> | <relocations> | Relocation entry |
| <reloc ptr> | <file_offsets> | File offset of relocation entries |
| <reloc strip> | <file_header> | True if relocation information was stripped from this file |
| <relocations> | <section> | Relocation entries |
| <return_address_register> | <fde> | Register used to pass the return address of this function |
| <row> | <table> | Table row |
| <section> | <dwarf> | DWARF section |

*Table 10−1. XML Tag Index*

| Tag Name | Context | Description |
|---|---|---|
| | <symbol> | Section containing the definition of this symbol |
| | <ti_coff> | COFF section |
| <section_count> | <file_header> | Number of section headers |
| <size_in_addrs> | <symbol> | Number of machine-address-sized units in function |
| <size_in_bits> | <symbol> | Size of symbol (bits) |
| <source> | <memory> | Source register |
| | <register> | Source register |
| <start_symbol> | <symbol> | First symbol in multi-symbol entity |
| <storage_class> | <symbol> | Storage class of this symbol |
| <storage_type> | <symbol> | Storage type of this symbol |
| <string> | <string_table> | String table entry |
| | <value> | String |
| <string_table> | <ti_coff> | String table |
| <string_table_size> | <string_table> | Size of string table |
| <sym_merge> | <file_header> | True if debug type-symbols were merged |
| <symbol> | <symbol_table> | Symbol table entry |
| <symbol_count> | <file_header> | Number of entries in the symbol table |
| <symbol_relative> | <reloc_entry> | Relocation is relative to the specified symbol |
| <symbol_table> | <ti_coff> | Symbol table |
| <table> | <fde> | FDE table |
| <tag> | <die> | Tag name |
| <tag index> | <symbol> | Reference to user-defined type |
| <target id> | <file_header> | Target ID; magic number identifying the target machine |
| <text> | <section> | True if this section contains code |
| <text_size> | <optional_file_header> | Size of executable code |

*Table 10−1. XML Tag Index*

| Tag Name | Context | Description |
|---|---|---|
| <text_start> | <optional_file_header> | Beginning address of executable code |
| <ti_coff> | <object_file> | TI COFF file |
| <tool_version> | <optional_file_header> | Tool version stamp |
| <type> | <attribute> | Attribute type |
| | <reloc_entry> | Type of relocation |
| <type_ref> | <value> | Type reference |
| <value> | <attribute> | Attribute value |
| | <reloc_entry> | Value |
| | <symbol> | Value |
| <vector> | <section> | True if this section contains a vector table (C55x only) |
| <version> | <compile_unit> | DWARF version |
| | <file_header> | Version ID; structure version of this COFF file |
| <virtual_addr> | <reloc_entry> | Virtual address to be relocated |
| | <section> | Virtual (load) address of section |
| <word_size> | <reloc_entry> | Number of address-sized units containing the relocation field |
| <xml_version> | <dwarf> | Version of the DWARF XML language |
| | <ti_coff> | Version of the COFF XML language |

## 10.3 Example XML Consumer

In this section, we present an example of a small application that uses the XML output of ofd55 to calculate the size of the executable code contained in an object file.

The example contains three source files: codesize.cpp, xml.h, and xml.cpp. When compiled into an executable named codesize, it can be used with ofd55 from the command line as follows:

```
% ofd55 –x a.out | codesize

Code Section Name: .text
Code Section Size: 44736

Code Section Name: .text2
Code Section Size: 64

Code Section Name: .text3
Code Section Size: 64

Total Code Size: 44864
```

### 10.3.1 The Main Application

The codesize.cpp file contains the main application for the object file display utility example.

```
//****************************************************************************
// CODESIZE.CPP – An example application that calculates the size of the      *
// executable code in an object file using the XML output                     *
// of the OFD utility.                                                        *
//****************************************************************************
#include ”xml.h”
#include <iostream>

using namespace std;

static void parse_XML_prolog(istream &in);

//****************************************************************************
// main() – List the names and sizes of the code sections (in octets), and    *
//          output the total code size.                                       *
//****************************************************************************
int main()
{
   //-------------------------------------------------------------------------
   // Build our tree of XML Entities from standard input (See xml.{cpp,h} for –
   // the definition of the XMLEntity object).                                –
   //-------------------------------------------------------------------------
   parse_XML_prolog(cin);
   XMLEntity *root = new XMLEntity(cin);
```

```
//---------------------------------------------------------------------------
// Fetch the XML Entities of the section roots. In other words, get a      -
// list of all the XMLEntity sub-trees named ”section” that are in the     -
// context of ”ofd->object_file->ti_coff”, where ”ofd” is the root of our  -
// XML document.                                                           -
//---------------------------------------------------------------------------
CEntityList    query_result;
const char *section_query[] =
   { ”ofd”, ”object_file”, ”ti_coff”, ”section”, NULL };

query_result = root->query(section_query);

//---------------------------------------------------------------------------
// Iterate over the section Entities, looking for code sections.           -
//---------------------------------------------------------------------------
CEntityList_CIt pit;
unsigned long total_code_size = 0;

for (pit = query_result.begin(); pit != query_result.end(); ++pit)
{

   //------------------------------------------------------------------------
   // Query for the name, text, and raw_data_size sub-entities of each     -
   // section. XMLEntity::query() always returns a list, even if there     -
   // will only ever be a maximum of one result. If the tag is not         -
   // found, an empty list is returned.                                    -
   //------------------------------------------------------------------------
   const char *section_name_query[] = { ”section”, ”name”,          NULL };
   const char *section_text_query[] = { ”section”, ”text”,          NULL };
   const char *section_size_query[] = { ”section”, ”raw_data_size”, NULL };

   CEntityList sname_l;
   CEntityList stext_l;
   CEntityList ssize_l;

   sname_l = (*pit)->query(section_name_query);
   stext_l = (*pit)->query(section_text_query);
   ssize_l = (*pit)->query(section_size_query);
   //------------------------------------------------------------------------
   // If a ”text” flag was found, this is a code section. Output           -
   // the section name and size, and add its size to our total code size   -
   // counter.                                                             -
   //------------------------------------------------------------------------
   if (stext_l.size() > 0)
   {
      unsigned long size;

      size = strtoul((*ssize_l.begin())->value().c_str(), NULL, 16);

      cout << ”Code Section Name:  ” << (*sname_l.begin())->value() << endl;
      cout << ”Code Section Size:  ” << size << endl;
      cout << endl;

      total_code_size += size;
   }
   }

   //-----------------------------------------------------------------------------
   // Output the total code size, and clean up. -
   //-----------------------------------------------------------------------------
   cout << ”Total Code Size: ” << total_code_size << endl;
   delete root;

   return 0;
}


//********************************************************************************
// parse_XML_prolog() - Parse the XML prolog, and throw it away. *
//********************************************************************************
static void parse_XML_prolog(istream &in)
{
   char c;

   while (true)
   {
      //--------------------------------------------------------------------------
      // Look for the next tag; if it is not an XML directive, we’re done.    -
      //--------------------------------------------------------------------------
      for (in.get(c); c != ’<’ && !in.eof(); in.get(c))
      ; // empty body

      if (in.eof()) return;
      if (in.peek() != ’?’) { in.unget(); return; }

      //--------------------------------------------------------------------------
      // Otherwise, read in the directive and continue. -
      //--------------------------------------------------------------------------
      for (in.get(c); c != ’>’ && !in.eof(); in.get(c))
         ; // empty body
   }
}
```

### 10.3.2 xml.h { Declaration of the XMLEntity Object

The xml.h file contains the declaration of the XMLEntity object for the codesize.cpp application.

```
//*****************************************************************************
// XML.H – Declaration of the XMLEntity object. *
//*****************************************************************************
#ifndef XML_H
#define XML_H
#include <list>
#include <string>


//*****************************************************************************
// Type Declarations. *
//*****************************************************************************
class XMLEntity;
typedef list<XMLEntity*>            EntityList;
typedef list<const XMLEntity*>      CEntityList;
typedef CEntityList::const_iterator CEntityList_CIt;
typedef EntityList::const_iterator  EntityList_CIt;
8
//*****************************************************************************
// CLASS XMLENTITY – A Simplified XML Entity Object.                          *
//*****************************************************************************
class XMLEntity
{
   public:
      XMLEntity  (istream &in, XMLEntity *parent=NULL);
      ~XMLEntity ();
      const CEntityList  query  (const char **context) const;
      const string      &tag    () const { return tag_m;      }
      const string      &value  () const { return value_m;    }

   private:
      void  parse_raw_tag (const string &raw_tag);
      void  sub_query     (CEntityList &result, const char **context) const;

      string      tag_m;       // Tag Name
      string      value_m;     // Value
      XMLEntity  *parent_m;    // Pointer to parent in XML hierarchy
      EntityList  children_m;  // List of children in XML hierarchy
};
#endif
```

### 10.3.3 xml.cpp { Definition of the XMLEntity Object

The xml.cpp file contains the definition of the XMLEntity object for the codesize.cpp application.

```
//****************************************************************************
// XML.CPP - Definition of the XMLEntity object. *
//****************************************************************************
#include ”xml.h”
#include <iostream>
#include <string>
#include <list>
#include <cstdlib>

//****************************************************************************
// XMLEntity::query() - Return the list of XMLEntities a list that reside *
// in the given XML context. *
//****************************************************************************
const CEntityList XMLEntity::query(const char **context) const
{
   CEntityList result;

   if (!*context) return result;

   sub_query(result, context);

   return result;
}

//****************************************************************************
// XMLEntity::sub_query() - Recurse through the XML tree looking for a match  *
//                          to the current query.                             *
//****************************************************************************
void XMLEntity::sub_query(CEntityList &result, const char **context) const
{
   if (!context[0] || tag() != context[0]) return;

   if (!context[1])
      result.push_front(this);
   else
   {
      EntityList_CIt pit;

      for (pit = children_m.begin(); pit != children_m.end(); ++pit)
         (*pit)->sub_query(result, context+1);
   }
   return;
}


//****************************************************************************
// XMLEntity::parse_raw_tag() - Cut out the tag name from the complete string *
// we found between the < > brackets. This throws out any attributes.        *
//****************************************************************************
void XMLEntity::parse_raw_tag(const string &raw_tag)
```

```
{
   string attribute;
   int    i;

   for (i = 0; i < raw_tag.size() && raw_tag[i] != ’ ’; ++i)
      tag_m += raw_tag[i];
}

//*****************************************************************************
// XMLEntity::XMLEntity() – Recursively construct a tree of XMLEntities from  *
//                          the given input stream.                           *
//*****************************************************************************
XMLEntity::XMLEntity(istream &in, XMLEntity *parent) :
tag_m(””), value_m(””), parent_m(parent)
{
   string raw_tag;
   char   c;
   int    i;
   //--------------------------------------------------------------------------
   // Read in the leading ’<’.                                                -
   //--------------------------------------------------------------------------
   in.get();

   //--------------------------------------------------------------------------
   // Store the tag name and attributes in ”raw_tag”, then call               -
   // process_raw_tag() to separate the tag name from the attributes and      -
   // store it in tag_m.                                                      -
   //--------------------------------------------------------------------------
   for (in.get(c); c != ’>’ && c != ’/’ && !in.eof(); in.get(c))
      raw_tag += c;

   parse_raw_tag(raw_tag);

   //--------------------------------------------------------------------------
   // If we’re reading in an end–tag, read in the closing ’>’ and return.     -
   //--------------------------------------------------------------------------
   if (c == ’/’) { in.get(c); return; }

   //--------------------------------------------------------------------------
   // Otherwise, parse our value.                                             -
   //--------------------------------------------------------------------------
   while (true)
   {
      //-----------------------------------------------------------------------
      // Read in the closing ’>’, then start reading in characters and add    -
      // them to value_m. Stop when we hit the beginning of a tag.            -
      //-----------------------------------------------------------------------
      for (in.get(c); c != ’<’; in.get(c)) value_m += c;

      //-----------------------------------------------------------------------
      // If we’re reading in a start tag, parse in the entire entity, and     -
      // add it to our child list (recursive constructor call).               -
      //-----------------------------------------------------------------------
      if (in.peek() != ’/’)
      {
         //-------------------------------------------------------------------
         // Put back the opening ’<’, since XMLEntity() expects to read it.   -
         //-------------------------------------------------------------------
      in.unget();
      children_m.push_front(new XMLEntity(in, this));
      }
      //----------------------------------------------------------------------
      // Otherwise, read in our end tag, and exit.                           -
      //----------------------------------------------------------------------
      else
      {
         for (in.get(c); c != ’>’; in.get(c))
            ; // empty body
         break;
      }
   }

   //-------------------------------------------------------------------------
   // Strip off leading and trailing white space from our value.             -
   //-------------------------------------------------------------------------
   for (i = 0; i < value_m.size(); i++)
      if (value_m[i] != ’ ’ && value_m[i] != ’\n’) break;
   value_m.erase(0, i);

   for (i = value_m.size()–1; i >= 0; i––)
      if (value_m[i] != ’ ’ && value_m[i] != ’\n’) break;
   value_m.erase(i+1, value_m.size()–i);
}

//****************************************************************************
// XMLEntity::~XMLEntity() – Delete a XMLEntity object.                      *
//****************************************************************************
XMLEntity::~XMLEntity()
{
   EntityList_CIt pit;

   for (pit = children_m.begin(); pit != children_m.end(); ++pit)
      delete (*pit);
}
```

## 10.4 Invoking the Name Utility

The name utility, *nm2000*, is used to print the list of names defined and referenced in a COFF object (.obj) or an executable file (.out). The value associated with the symbol and an indication of the kind of symbol is also printed.

To invoke the name utility, enter the following:

**nm2000** [−*options*] [*input filename*]

| | |
|---|---|
| **nm2000** | is the command that invokes the name utility. |
| *input filename* | is a COFF object file (.obj), an executable file (.out), or an archive file. For an archive file, the name utility processes each object file in the archive. |
| *options* | identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (−). The name utility options are as follows: |

| | |
|---|---|
| **−a** | prints all symbols. |
| **−c** | also prints C_NULL symbols. |
| **−d** | also prints debug symbols. |
| **−f** | prepends file name to each symbol. |
| **−g** | prints only global symbols. |
| **−h** | shows the current help screen. |
| **−l** | produces a detailed listing of the symbol information. |
| **−n** | sorts symbols numerically rather than alphabetically. |
| **−o***file* | outputs to the given file. |
| **−p** | causes the name utility to not sort any symbols. |
| **−q** | (quiet mode) suppresses the banner and all progress information. |
| **−r** | sorts symbols in reverse order. |
| **−t** | also prints tag information symbols. |
| **−u** | only prints undefined symbols. |

## 10.5 Invoking the Strip Utility

The strip utility, *strip2000*, is used to remove symbol table and debugging information from object and executable files.

To invoke the strip utility, enter the following:

```
strip2000 [−p] input filename [input filename]
```

| | |
|---|---|
| **strip2000** | is the command that invokes the strip utility. |
| *input filename* | is a COFF object file (.obj) or an executable file (.out). |
| *options* | identifies the strip utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (−). The strip utility option follows: |

| | |
|---|---|
| **−p** | removes all information not required for execution. This option causes more information to be removed than the default behavior, but the object file is left in a state that cannot be linked. This option should be used only with executable (.out) files. |

When the strip utility is invoked, the input object files are replaced with the stripped version.

Chapter 11

# Hex-Conversion Utility Description

The TMS320C28x™ 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

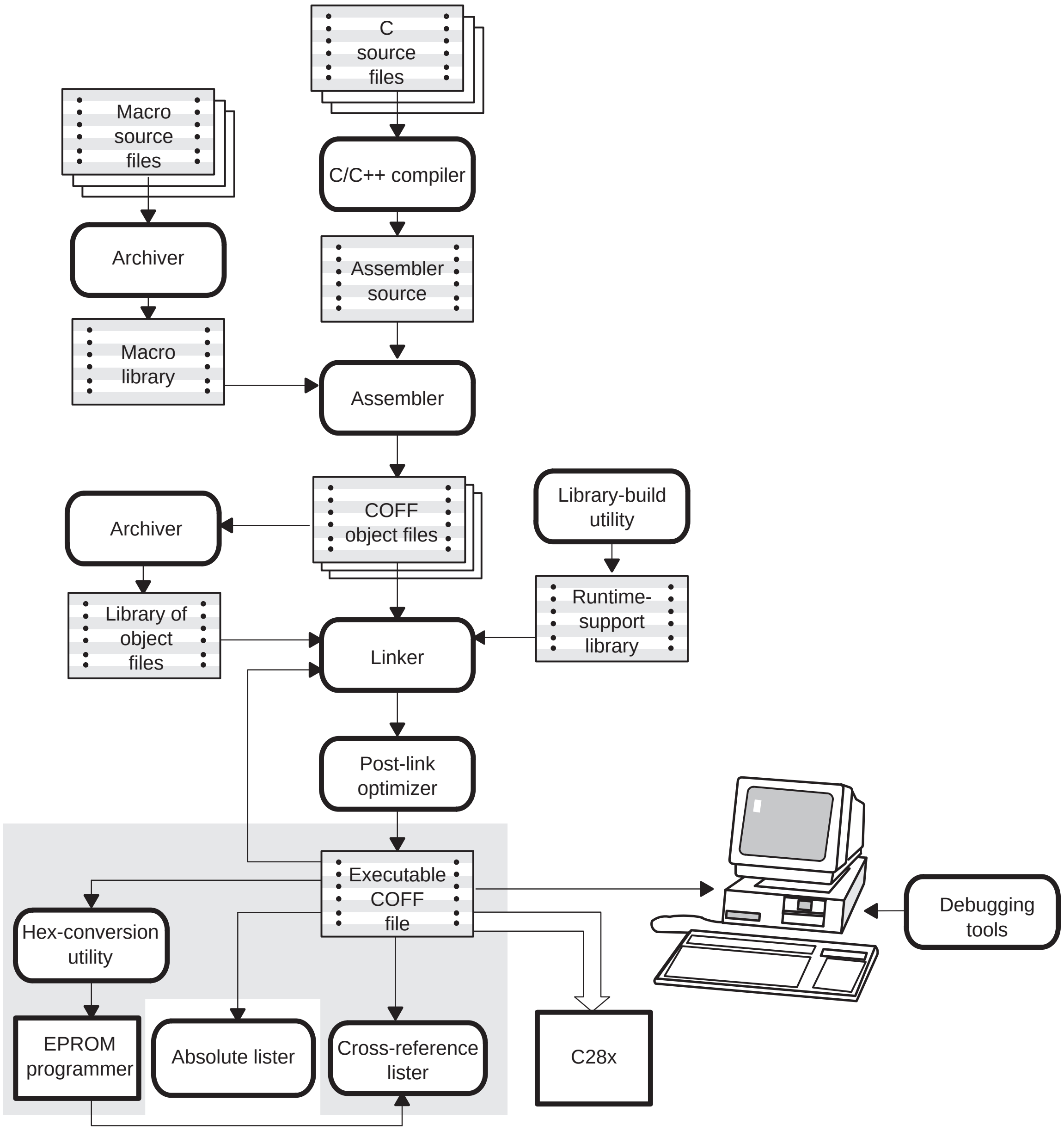
**Topic** **Page**

**11.1 The Hex-Conversion Utility's Role in the Software Development Flow . . . 11-2**
**11.2 Invoking the Hex-Conversion Utility . . . 11-3**
**11.3 Understanding Memory Widths . . . 11-8**
**11.4 The ROMS Directive . . . 11-15**
**11.5 The SECTIONS Directive . . . 11-21**
**11.6 Excluding a Specified Section . . . 11-23**
**11.7 Assigning Output Filenames . . . 11-24**
**11.8 Image Mode and the −fill Option . . . 11-26**
**11.9 Building a Table for an On-Chip Boot Loader . . . 11-28**
**11.10 Controlling the ROM Device Address . . . 11-38**
**11.11 Object Formats . . . 11-39**
**11.12 Hex-Conversion Utility Error Messages . . . 11-45**

## 11.1 The Hex-Conversion Utility's Role in the Software Development Flow

Figure 11−1 highlights the role of the hex-conversion utility in the software development process.

*Figure 11−1. The Hex-Conversion Utility in the TMS320C28x Software Development Flow*



## 11.2 Invoking the Hex-Conversion Utility

There are two basic methods for invoking the hex-conversion utility:

- **Specify the options and filenames on the command line.** The following example converts the file firmware.out into TI-Tagged format, producing two output files, firm.lsb and firm.msb.

  ```
  hex2000 –t firmware –o firm.lsb –o firm.msb
  ```

- **Specify the options and filenames in a command file.** You can create a batch file that stores command line options and filenames for invoking the hex-conversion utility. The following example invokes the utility using a command file called hexutil.cmd:

  ```
  hex2000 hexutil.cmd
  ```

  In addition to regular command line information, you can use the hex-conversion utility ROMS and SECTIONS directives in a command file.

### 11.2.1 Invoking the Hex-Conversion Utility From the Command Line

To invoke the hex-conversion utility, enter:

**hex2000** [*options*] *filename*

| | |
|---|---|
| **hex2000** | is the command that invokes the hex-conversion utility. |
| *options* | supplies additional information that controls the hex-conversion process. You can use options on the command line or in a command file.<br>❑ All options are preceded by a hyphen and are not case sensitive.<br>❑ Several options have an additional parameter that must be separated from the option by at least one space.<br>❑ Options with multicharacter names must be spelled exactly as shown in this document; no abbreviations are allowed.<br>❑ 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 11.2.2, *Invoking the Hex-Conversion Utility With a Command File*, on page 11-6). |

*Table 11−1. Hex Conversion Utility Options*

*(a) General options*

The general options control the overall operation of the hex conversion utility.

| **Option** | **Description** | **Page** |
|---|---|---|
| −exclude *section* | Ignore specified section | 11-23 |
| −map *filename* | Generate a map file | 11-19 |
| −o *filename* | Specify an output filename | 11-24 |
| −q | Run quietly (when used, it must appear *before* other options) | 11-6 |

*(b) Image options*

The image options create a continuous image of a range of target memory.

| **Option** | **Description** | **Page** |
|---|---|---|
| −fill *value* | Fill holes with *value* | 11-27 |
| −image | Specify image mode | 11-26 |
| −zero | Reset the address origin to zero in image mode | 11-38 |

*(c) Memory options*

The memory options configure the memory widths for your output files.

| **Option** | **Description** | **Page** |
|---|---|---|
| −memwidth *value* | Define the system memory word width (default 32 bits) | 11-9 |
| −romwidth *value* | Specify the ROM device width (default depends on format used) | 11-11 |

*Table 11−1. Hex Conversion Utility Options (Continued)*

*(d) Output formats*

The output formats specify the format of the output file.

| Option | Description | Page |
|---|---|---|
| −a | Select ASCII-Hex | 11-40 |
| −i | Select Intel | 11-41 |
| −m | Select Motorola-S | 11-42 |
| −t | Select TI-Tagged | 11-43 |
| −x | Select Tektronix | 11-44 |

*(e) Boot-loader options for all C28x devices*

The boot-loader options for all C28x devices control how the hex conversion utility builds the boot table.

| Option | Description | Page |
|---|---|---|
| −boot | Convert all sections into bootable form (use instead of a SECTIONS directive) | 11-28, 11-30 |
| −bootorg *value* | Specify the source address of the boot loader table | 11-30 |
| −e *value* | Specify the entry point at which to begin execution after boot loading. | 11-31 |
| −gpio8 | Specify table source as the GP I/O port, 8-bit mode | 11-31 |
| −gpio16 | Specify table source as the GP I/O port, 16-bit mode | 11-31 |
| −lospcp *value* | Specify the initial value for the LOSPCP register | 11-31 |
| −sci8 | Specify table source as the SCI-A port, 8-bit mode | 11-31 |
| −spi8 | Specify table source as the SPI-A port, 8-bit mode | 11-31 |
| −spibrr *value* | Specify the initial value for the SPIBRR register | 11-31 |

### 11.2.2 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 11.4, *The ROMS Directive*, on page 11-15.)
- **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 7-29.)
- **Comments.** You can add comments to your command file by using the /* and */ delimiters. For example:

  ```
  /*   This is a comment.   */
  ```

To invoke the utility and use the options you defined in a command file, enter:

**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:
}
```

## 11.3 Understanding Memory Widths

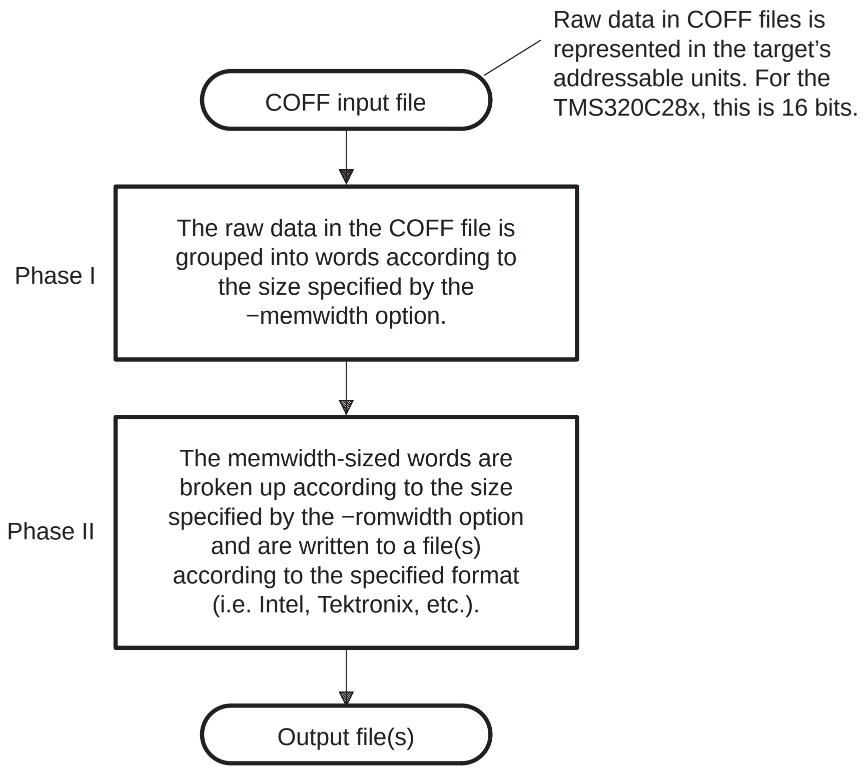
The hex-conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex-conversion utility, *you must understand how the utility treats word widths*. Three widths are important in the conversion process:

- ❑ Target width
- ❑ Memory width
- ❑ ROM width

The terms *target word*, *memory word*, and *ROM word* refer to a word of target, memory, and ROM width, respectively.

Figure 11−2 illustrates the two separate and distinct phases of the hex-conversion utility's process flow.

*Figure 11−2. Hex-Conversion Utility Process Flow*


Raw data in COFF files is represented in the target's addressable units. For the TMS320C28x, this is 16 bits.
COFF input file
Phase I
The raw data in the COFF file is grouped into words according to the size specified by the −memwidth option.
Phase II
The memwidth-sized words are broken up according to the size specified by the −romwidth option and are written to a file(s) according to the specified format (i.e. Intel, Tektronix, etc.).
Output file(s)

### 11.3.1 Target Width

Target width is the unit size (in bits) of the target processor’s word. The unit size corresponds to the data bus size on the target processor. The width is fixed for each target and cannot be changed. The TMS320C28x targets have a width of 16 bits.

### 11.3.2 Specifying the Memory Width

Memory width is the physical width (in bits) of the memory system. Usually, the memory system is physically the same width as the target processor width: a 16-bit processor has a 16-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, 16 bits).

You can change the memory width by:

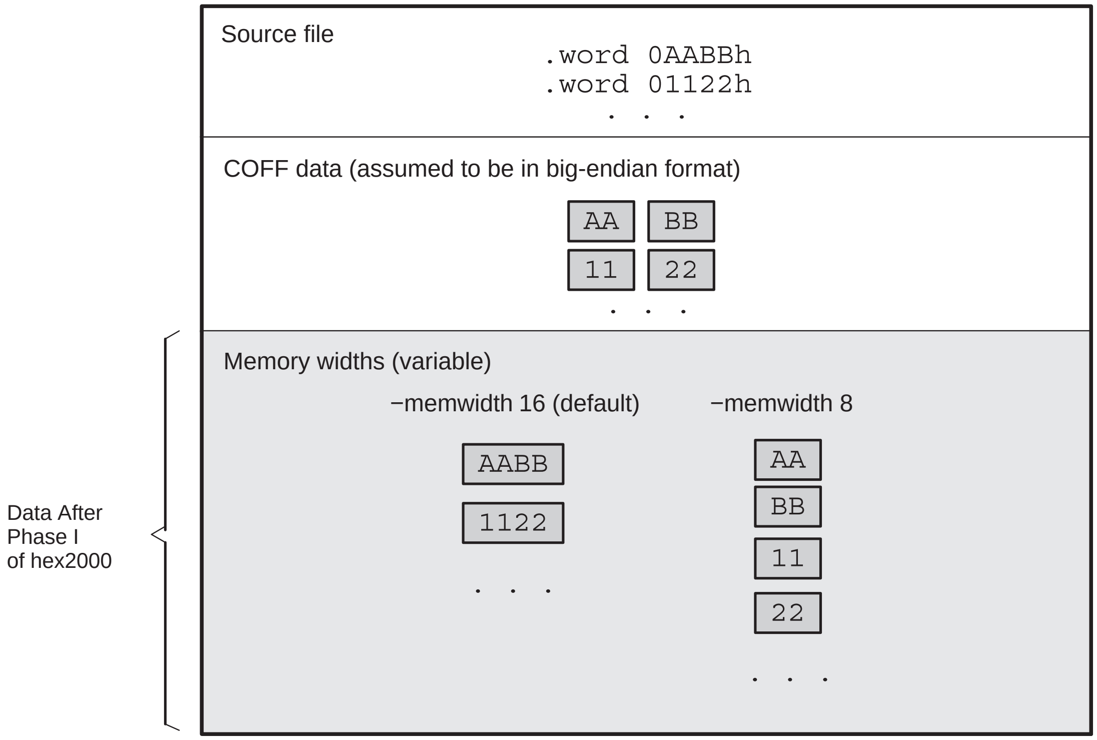
- Using the **−memwidth** option. This changes the memory-width value for the entire file.
- Setting the **memwidth** parameter of the ROMS directive. This changes the memory-width value for the address range specified in the ROMS directive and overrides the −memwidth option for that range. See section 11.4, *The ROMS Directive*, on page 11-15.

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 11−3 demonstrates how the memory width is related to COFF data.

*Figure 11−3.COFF Data and Memory Widths*



### 11.3.3 Partitioning Data Into Output Files

ROM width specifies the physical width (in bits) of each ROM device and corresponding output file (usually one byte or eight bits). The ROM width determines how the hex-conversion utility partitions the data into output files. After the 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 $\geq$ ROM width:

  number of files = memory width $\div$ 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 11.4, *The ROMS Directive*, on page 11-15.

For both methods, use a value that is a power of 2 greater than or equal to 8.

If you select a ROM width that is wider than the natural size of the output format (16 bits for TI-Tagged or 8 bits for all others), the utility simply writes multibyte fields into the file.

Figure 11−4 illustrates how the 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:

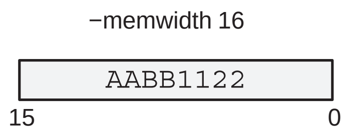
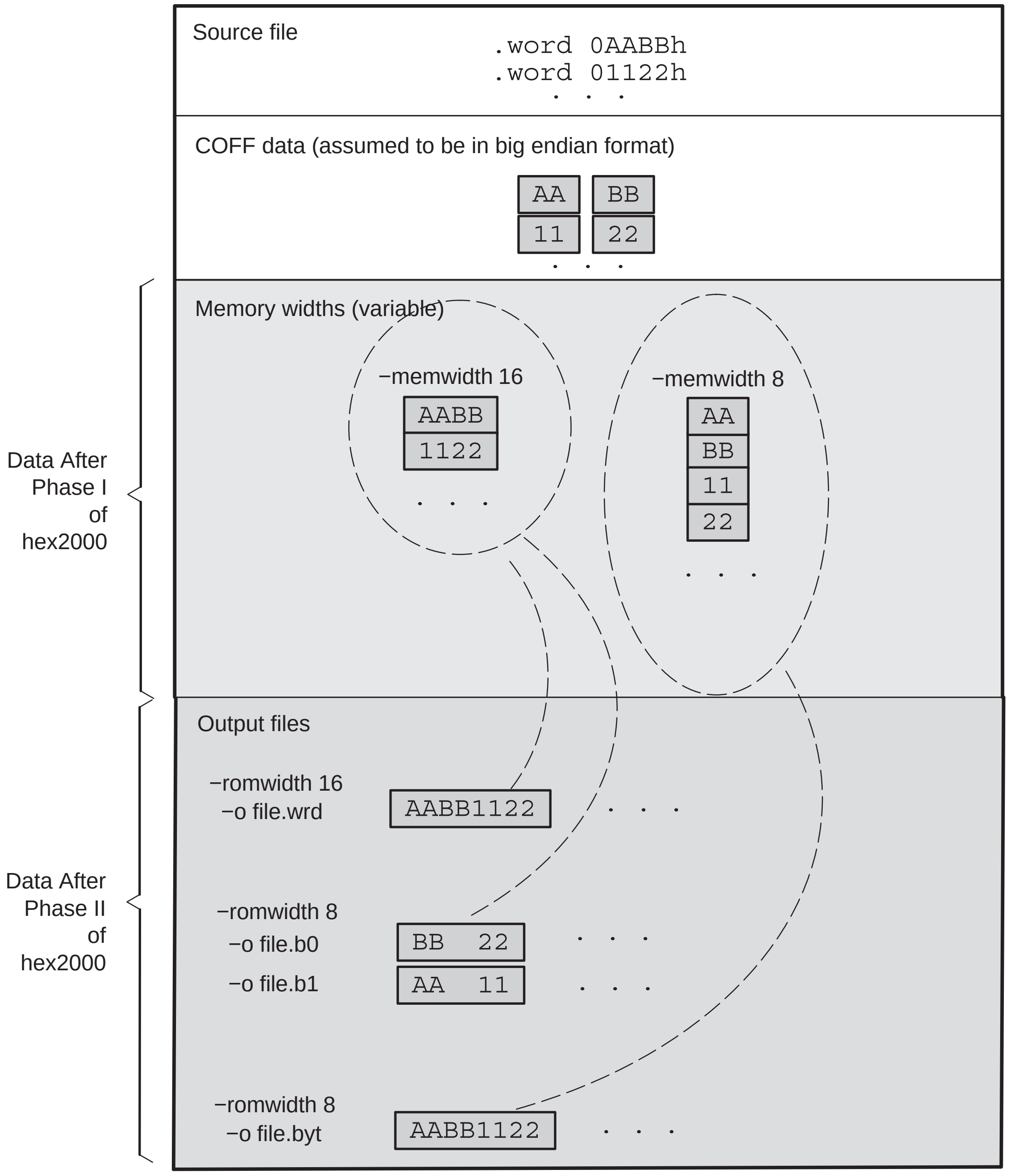


*Figure 11−4. Data, Memory, and ROM Widths*



### 11.3.4 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

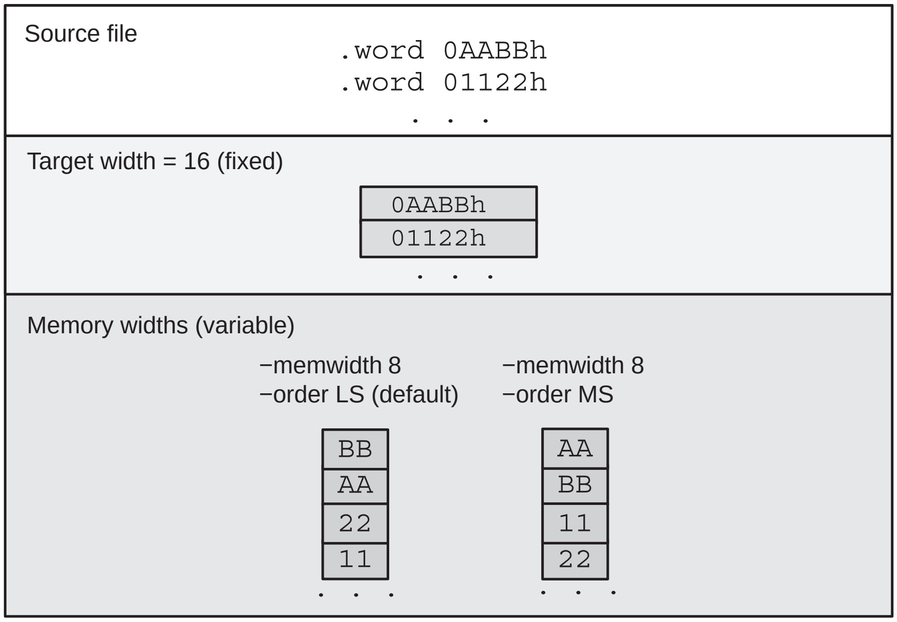
By default, the utility uses little-endian format because the TMS320C28x boot loaders expect the data in this order. Unless you are using your own boot loader program, avoid using −order MS.

**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 11−5 demonstrates how −order affects the conversion process. This figure, and the previous figure, Figure 11−4, explain the condition of the data in the hex conversion utility output files.

*Figure 11−5. Varying the Word Order*



## 11.4 The ROMS Directive

The ROMS directive specifies the physical memory configuration of your system as a list of address-range parameters.

Each address range produces one set of files containing the hex-conversion utility output data that corresponds to that address range. Each file can be used to program one single ROM device.

The ROMS directive is similar to the MEMORY directive of the TMS320C28x 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 11.3.3, *Partitioning Data Into Output Files*, on page 11-10). Any value you specify here overrides the −romwidth option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8. |
| **memwidth** | specifies the memory width of the range in bits (see section 11.3.2, *Specifying the Memory Width*, on page 11-9). 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 11.5, *The SECTIONS Directive*, on page 11-21.) |
| **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.<br><br>The fill value must be a decimal, octal, or hexadecimal constant with a width equal to the target width. Any value you specify here overrides the −fill option. When using fill, you must also use the −image command line option. See section 11.8.2, *Specifying a Fill Value*, on page 11-27. |
| **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.<br><br>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 11.3.3, *Partitioning Data Into Output Files*, on page 11-10. The utility warns you if you list too many or too few filenames. |

Unless you are using the −image option, all of the parameters that define a range are optional; the commas and equal signs are also optional. A range with no origin or length defines the entire address space. In image mode, an origin and length are required for all ranges.

Ranges must not overlap and must be listed in order of ascending address.

### 11.4.1 When to Use the ROMS Directive

If you do not use a ROMS directive, the utility defines a single default range that includes the entire address space. This is equivalent to a ROMS directive with a single range without origin or length.

Use the ROMS directive when you want to:

- **Program large amounts of data into fixed-size ROMs**. When you specify memory ranges corresponding to the length of your ROMs, the utility automatically breaks the output into blocks that fit into the ROMs.
- **Restrict output to certain segments**. You can also use the ROMS directive to restrict the conversion to a certain segment or segments of the target address space. The utility does not convert the data that falls outside of the ranges defined by the ROMS directive. Sections can span range boundaries; the utility splits them at the boundary into multiple ranges. If a section falls completely outside any of the ranges you define, the utility does not convert that section and issues no messages or warnings. 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.

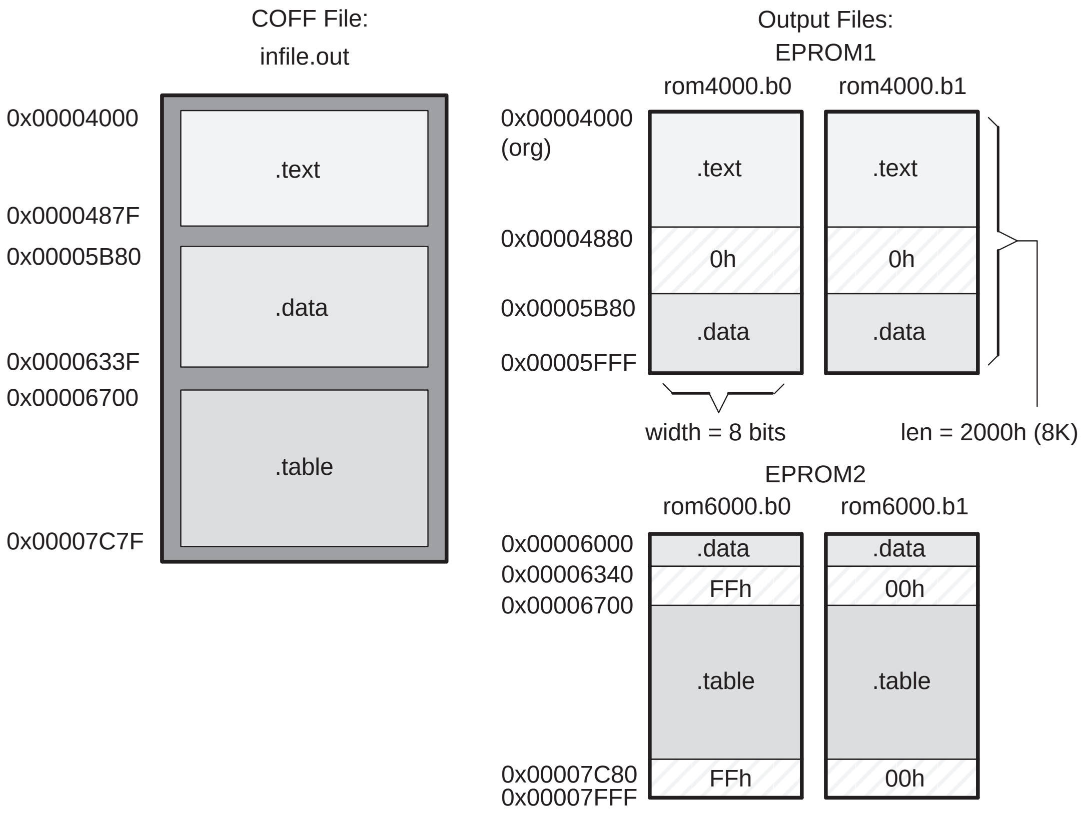
### 11.4.2 An Example of the ROMS Directive

The ROMS directive in Example 11−1 shows how 16K bytes of 16-bit memory could be partitioned for two 8K $\times$ 8-bit EPROMs. Figure 11−6 illustrates the input and output files.

*Example 11−1. A ROMS Directive Example*

```
infile.out
−image
−memwidth 16

ROMS
{
   EPROM1: org = 0x00004000, len = 0x2000, romwidth = 8
           files = { rom4000.b0, rom4000.b1}

   EPROM2: org = 0x00006000, len = 0x2000, romwidth = 8,
           fill = 0xFF00,
           files = { rom6000.b0, rom6000.b1}
}
```

*Figure 11−6. The infile.out File Partitioned Into Four Output Files*



The map file (specified with the −map option) is advantageous when you use the ROMS directive with multiple ranges. The map file shows each range, its parameters, names of associated output files, and a list of contents (section names and fill values) broken down by address. Example 11−2 is a segment of the map file resulting from Example 11−1.

*Example 11−2. Map File Output From Example 11−1 Showing Memory Ranges*

```
-----------------------------------------------------
00004000..00005fff  Page=0  Width=8  ”EPROM1”
-----------------------------------------------------
   OUTPUT FILES:   rom4000.b0   [b0..b7]
                   rom4000.b1   [b8..b15]

   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
-----------------------------------------------------
00006000..00007fff  Page=0  Width=8  ”EPROM2”
-----------------------------------------------------
   OUTPUT FILES:   rom6000.b0   [b0..b7]
                   rom6000.b1   [b8..b15]

   CONTENTS: 00006000..0000633f .data
             00006340..000066ff FILL = 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.

## 11.5 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 TMS320C28x 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 TMS320C28x 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 11.2.2, *Invoking the Hex-Conversion Utility With a Command File*, on page 11-6.) The general syntax for the SECTIONS directive is:

```
SECTIONS
{
   sname: [paddr=value],
   sname: [paddr=boot]
   sname: [= boot ],
   ...
}
```

| | |
|---|---|
| **SECTIONS** | begins the directive definition. |
| *sname* | identifies a section in the COFF input file. If you specify a section that 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. It can also be the word **boot** (to indicate a boot table section for use with the on-chip boot loader). *If your file contains multiple sections, and if one section uses a paddr parameter, then all sections must use a paddr parameter.* |
| **= boot** | configures a section for loading by the on-chip boot loader. This is equivalent to using **paddr=boot**. Boot sections have a physical address determined both by the target processor type and by the various boot-loader-specific command line options. |

The commas 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 }
```

## 11.6 Excluding a Specified Section

The −exclude *section* option can be used to inform the hex utility to ignore the specified section. If a SECTIONS directive is used, it overrides the −exclude option.

For example, if a SECTIONS directive containing the section name *mysect* is used and an −exclude *mysect* is specified, the SECTIONS directive takes precedence and *mysect* is not excluded.

The −exclude option has a limited wildcard capability. The * character can be placed at the beginning or end of the name specifier to indicate a suffix or prefix, respectively. For example, −exclude sect* disqualifies all sections that begin with the characters sect.

If you specify the −exclude option on the command line with the * wildcard, enter quotes around the section name and wildcard. For example, −exclude”sect*”. Using quotes prevents the * form being interpreted by the hex conversion utility. If −exclude is in a command file, then the quotes should not be specified.

## 11.7 Assigning Output Filenames

When the hex-conversion utility translates your 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 |

## 11.8 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.

### 11.8.1 Generating a Memory Image

With the −image option, the utility generates a memory image by completely filling all of the mapped ranges specified in the ROMS directive.

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.

### 11.8.2 Specifying a Fill Value

The −fill option specifies a value for filling the holes between sections. The fill value must be specified as an integer constant following the −fill option. The width of the constant is assumed to be that of a word on the target processor. For example, specifying −fill 0FFh results in a fill pattern of 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.

### 11.8.3 Steps to Follow in Image Mode

**Step 1:** Define the ranges of target memory with a ROMS directive. See section 11.4, *The ROMS Directive*, on page 11-15 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.

## 11.9 Building a Table for an On-Chip Boot Loader

Some C28x devices, such as the F2810/12, have a built-in boot loader that initializes memory with one or more blocks of code or data. The boot loader uses a special table (a *boot table*) stored in memory (such as EPROM) or loaded from a device peripheral (such as a serial or communications port) to initialize the code or data. The hex conversion utility supports the boot loader by automatically building the boot table.

### 11.9.1 Description of the Boot Table

The input for a boot loader is the boot table. The boot table contains records that instruct the on-chip loader to copy blocks of data contained in the table to specified destination addresses. The boot table can be stored in memory or read in through a device peripheral.

The hex conversion utility automatically builds the boot table for the boot loader. Using the utility, you specify the COFF sections you want the boot loader to initialize and the table location. The hex conversion utility builds a complete image of the table according to the format specified and converts it into hexadecimal in the output files. Then, you can burn the table into ROM or load it by other means.

The boot loader supports loading from memory that is narrower than the normal width of memory. For example, you can boot a 16-bit TMS320C28x from a single 8-bit EPROM by using the −memwidth option to configure the width of the boot table. The hex conversion utility automatically adjusts the table’s format and length. See the boot loader example in the *TMS320C28x DSP CPU and Instruction Set Reference Guide* for an illustration of a boot table.

### 11.9.2 The Boot Table Format

The boot table format is simple. Typically, there is a header record containing a key value that indicates memory width, entry point, and values for control registers. Each subsequent block has a header containing the size and destination address of the block followed by data for the block. Multiple blocks can be entered. The table ends with a header containing size zero. See the boot loader section in the *TMS320C28x DSP CPU and Instruction Set Reference Guide* for more information.

### 11.9.3 Controlling the Boot Loader

Table 11−2 summarizes the hex conversion utility options available for the boot loader.

*Table 11−2. Boot-Loader Options*

| Option | Description |
|---|---|
| −boot | Convert all input sections into bootable form (use instead of a SECTIONS directive) |
| −bootorg *value* | Specify the source address of the boot loader table |
| −e *value* | Specify the entry point at which to begin execution after boot loading. The *value* can be an address or a global symbol. |
| −gpio8 | Specify the source of the boot loader table as the GP I/O port, 8-bit mode |
| −gpio16 | Specify the source of the boot loader table as the GP I/O port, 16-bit mode |
| −lospcp *value* | Specify the initial value for the LOSPCP register. This value is used only for the spi8 boot table format and is ignored for all other formats. A value greater than 0x7F is truncated to 0x7F. |
| −sci8 | Specify the source of the boot loader table as the SCI-A port, 8-bit mode |
| −spi8 | Specify the source of the boot loader table as the SPI-A port, 8-bit mode. This format initializes the LOSPCP and SPIBRR registers. If initial values for these registers were not specified, the hex conversion utility used 0x2 and 0x7F respectively. |
| −spibrr *value* | Specify the initial value for the SPIBRR register. This value is used only for the spi8 boot table format and is ignored for all other formats. A value greater than 0x7F is truncated to 0x7F. |

#### 11.9.3.1 Building the Boot Table

To build the boot table, follow these steps:

**Step 1:** **Link the file**. Each block of the boot table data corresponds to an initialized section in the COFF file. Uninitialized sections are not converted by the hex conversion utility (see Section 11.5, *The SECTIONS Directive*, on page 11-21).

When you select a section for placement in a boot-loader table, the hex conversion utility places the section's *load address* in the destination address field for the block in the boot table. The section content is then treated as raw data for that block.

*The hex conversion utility does not use the section run address.* When linking, you need not worry about the ROM address or the construction of the boot table—the hex conversion utility handles this.

**Step 2:** **Identify the bootable sections**. You can use the −boot option to tell the hex conversion utility to configure all sections for boot loading. Or, you can use a SECTIONS directive to select specific sections to be configured. If you use a SECTIONS directive, −boot is ignored.

**Step 3:** **Set the boot table format.** Specify the −gpio8, −gpio16, −sci8, or −spi8 option to set the source of the boot table. You do not need to specify the memwidth and romwidth as the utility will set these formats automatically. If −memwidth and −romwidth are used after a format option, they override the default for the format.

**Step 4:** **Set the ROM address of the boot table**. Use the −bootorg option to set the source address of the complete table. For example, if you are using the C28x and booting from memory location 0x3FF000, specify −bootorg 0x3FF000. The address field in the hex conversion utility output file will then start at 0x3FF000.

**Step 5:** **Set boot-loader-specific options.** Set entry point and control register values as needed.

**Step 6:** **Describe your system memory configuration**. See Section 11.3, *Understanding Memory Widths*, on page 11-8 and Section 11.4, *The ROMS Directive*, on page 11-15 for details.

#### *11.9.3.2 Leaving Room for the Boot Table*

The complete boot table is similar to a single section containing all of the header records and data for the boot loader. The address of this section is the boot table origin. As part of the normal conversion process, the hex conversion utility converts the boot table to hexadecimal format and maps it into the output files like any other section.

Be sure to leave room in your system memory for the boot table, especially when you are using the ROMS directive. The boot table cannot overlap other nonboot sections or unconfigured memory. Usually, this is not a problem; typically, a portion of memory in your system is reserved for the boot table. Simply configure this memory as one or more ranges in the ROMS directive, and use the −bootorg option to specify the starting address.

### 11.9.4 Booting From a Device Peripheral

You can choose to boot from the F2810/12 serial or parallel port by using the −gpio8, −gpio16, −sci8, or −spi8 boot table format option. The initial value for the LOSPCP register can be specified with the −lospcp option. The initial value for the SPIBRR register can be specified with the −spibrr option. Only the −spi8 format uses these control register values in the boot table.

If the register values are not specified for the −spi8 format, the hex conversion utility uses the default value 0x02 for LOSPCP and 0x7F for SPIBRR. When the boot table format options are specified and the ROMS directive is not specified, the ASCII format hex utility output does not produce the address record.

### 11.9.5 Setting the Entry Point for the Boot Table

After completing the boot load process, execution starts at the default entry point specified by the linker and contained in the COFF file. By using the −e option with the hex conversion utility, you can set the entry point to a different address.

For example, if you want your program to start running at address 0x0123 after loading, specify −e 0x0123 on the command line or in a command file. You can determine the −e address by looking at the map file that the linker generates.

**Note: Valid Entry Points**

The value can be a constant, or it can be a symbol that is externally defined (for example, with a .global) in the assembly source.

### 11.9.6 Using the C28x Boot Loader

This subsection explains how to use the hex conversion utility with the boot loader for C28x devices. The C28x boot loader accepts the formats listed in Table 11−3.

*Table 11−3. Boot Table Source Formats*

| Format | Option |
|---|---|
| Parallel boot GP I/O 8-bit | −gpio8 |
| Parallel boot GP I/O 16-bit | −gpio16 |
| 8-bit SCI boot | −sci8 |
| 8-bit SPI boot | −spi8 |

The F2810/12 can boot through the SCI-A 8-bit, SPI-A 8-bit, GP I/O 8-bit, or GP I/O 16-bit interface. The format of the boot table is shown in Table 11−4.

*Table 11−4. Boot Table Format*

| Description | Word | Content |
|---|---|---|
| Boot table header | 1 | Key value (0x10AA or 0x08AA) |
| | 2−9 | Register initialization value or reserved for future use |
| | 10−11 | Entry point |
| Block header | 12 | Block size in number of words (n1) |
| | 13−14 | Destination address of the block |
| Block data | 15 | Raw data for the block (n1 words) |
| Block header | 16 + nl | Block size in number of words |
| | . | Destination address of the block |
| Block data | . | Raw data for the block |
| Additional block headers and data, as required | . . . | Content as appropriate |
| Block header with size 0 | | 0x0000; indicates the end of the boot table |

The C28x can boot through either the serial 8-bit or parallel interface with either 8- or 16-bit data. The format is the same for any combination: the boot table consists of a field containing the destination address, a field containing the length, and a block containing the data. You can boot only one section. If you are booting from an 8-bit channel, 16-bit words are stored in the table with MSBs first; the hex conversion utility automatically builds the table in the correct format. Use the following options to specify the boot table source:

❑ To boot from a SCI-A port, specify −spi8 when invoking the utility. Do not specify −memwidth or −romwidth.

❑ To boot from a SPI-A port, specify −sci8 when invoking the utility.  Do not specify −memwidth or −romwidth. Use −lospcp to set the initial value for the LOSPCP register and −spibrr to set the initial value for the SPIBRR register. If the register values are not specified for the −spi8 format, the hex conversion utility uses the default value 0x02 for LOSPCP and 0x7F for SPIBRR.

❑ To load from a general-purpose parallel I/O port, invoke the utility with −gpio8 or −gpio16.  Do not specify −memwidth or −romwidth.

The command file in Example 11−3 allows you to boot the .text and .cinit sections of test.out from a 16-bit-wide EPROM at location 0x3FFC00. The map file test.map is also generated.

*Example 11−3.  Sample Command File for Booting From 8-Bit SPI Boot*

```
/*--------------------------------------------------------------------*/
/* Hex convertor command file.                                        */
/*--------------------------------------------------------------------*/
test.out            /* Input COFF file */
-a                  /* Select ASCII format */
-map test.map       /* Specify the map file */
-o test_spi8.hex    /* Hex utility out file */
-boot               /* Consider all the input section as boot sections */
-spi8               /* Specify the SPI 8-bit boot format */
-lospcp 0x3F        /* Set the initial value for the LOSPCP as 0x3F*/
                    /* -spibrr option is not specified to show that the*/
                    /* hex utility uses the default value (0x7F)       */
-e 0x3F0000         /* Set the entry point */
```

The command file in Example 11−3 generates the out file in Example 11−4. The control register values are coded in the boot table header and the boot table header has the address that is specified with the −e option.

*Example 11−4. Sample Hex Convertor Out File for Booting From 8-Bit SPI Boot*

```
Key value
  |   LOSPCP initial value
  |   |  SPIBRR register initial value
  |   |  |                    Reserved for future use
  |   |  |                    |                         Entry point
  |   |  |                    |                         |
----- -- -- ----------------------------------------- -----------
AA 08 3F 7F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3F 00 00 00 90 00
                                                                  -----
                                                                    |
                                       Length of first block in words

    Address of the first block
    |
-----------
3F 00 00 00 42 B8 00 9A 04 28 05 00 06 00 AD 28 88 10 69 FF 1F 56 16 56
1A 56 40 29 1F 76 00 00 02 29 1B 76 22 76 A9 28 90 00 A8 28 3F 00 01 09
1D 61 FF 76 90 00 04 29 0F 6F 00 9B A9 24 01 DF 04 6C 04 29 A8 24 01 DF
A6 1E A1 F7 86 24 A7 06 A1 81 01 09 A7 1E A9 24 03 63 5C FF 04 3B A9 59
00 77 00 77 01 DF 09 00 EA FF 1A 76 A9 28 FF FF A8 28 FF FF 01 09 0E 61
FF 76 FF FF 06 6F 01 DF BD C3 A7 1E 67 3E BE C5 A9 24 01 DF A8 24 58 FF
F7 60 7F 76 00 00 7F 76 4B 00 BD B2 42 B8 BD AA 02 C5 67 3E 40 B8 00 59
A1 92 0D EC 03 56 A1 01 A9 08 40 10 A9 5A 82 DA C2 C5 67 3E A1 92 FF 9C
A9 59 FA ED 40 B8 02 06 03 EC A7 1E 67 3E 40 B8 04 06 03 EC A7 1E 67 3E
00 77 00 6F 42 B8 BD B2 02 C5 A4 8B 67 3E 40 B8 00 92 20 52 06 64 42 B8
00 C5 67 3E 01 9A 0D 6F 00 93 00 0A 03 56 A8 01 A9 5C A4 08 40 10 42 B8
C4 B2 00 C5 67 3E 00 9A BE 8B 06 00 00 6F 06 00 42 B8 02 A8 06 00 42 B8
00 A8 06 00
Length of second block in words
  |       Address of the second block
  |       |
----- -----------
1A 00 3F 00 90 00 04 00 84 10 01 00 02 00 03 00 04 00 01 00 00 10 00 00
02 00 02 10 00 00 00 00 02 00 04 10 00 00 00 00 02 00 80 10 89 00 3F 00
02 00 82 10 89 00 3F 00 00 00 00 00
                              -----
                                |
                   Terminating header with length zero
```

The command file in Example 11−5 allows you to boot the .text and .cinit sections of test.out from the 16-bit parallel GP I/O port. The map file test.map is also generated.

*Example 11−5. Sample Command File for C28x 16-Bit Parallel Boot GP I/O*

```
/*----------------------------------------------------------------------*/
/* Hex convertor command file.                                         */
/*----------------------------------------------------------------------*/
test.out            /* Input COFF file */
-a                  /* Select ASCII format */
-map test.map       /* Specify the map file */
-o test_gpio16.hex  /* Hex utility out file */
-gpio16             /* Specify the 16-bit GP I/O boot format */

SECTIONS
{
   .text: paddr=BOOT
   .cinit: paddr=BOOT
}
```

The command file in Example 11−5 generates the out file in Example 11−6.

*Example 11−6. Sample Hex Convertor Out File for C28x 16-Bit Parallel Boot GP I/O*

```
  Key value
  |                       Reserved for future use
  |                       |                              Entry point
  |                       |                              |
----- ------------------------------------------------ -----------
10 AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3F 00 05 00 90
                                                                  -----
                                                                    |
                                       Length of first block in words
    Address of the first block
    |
-----------
00 3F 00 00 B8 42 9A 00 28 04 00 05 00 06 28 AD 10 88 FF 69 56 1F 56 16
56 1A 29 40 76 1F 00 00 29 02 76 1B 76 22 28 A9 00 90 28 A8 00 3F 09 01
61 1D 76 FF 00 90 29 04 6F 0F 9B 00 24 A9 DF 01 6C 04 29 04 24 A8 DF 01
1E A6 F7 A1 24 86 06 A7 81 A1 09 01 1E A7 24 A9 63 03 FF 5C 3B 04 59 A9
77 00 77 00 DF 01 00 09 FF EA 76 1A 28 A9 FF FF 28 A8 FF FF 09 01 61 0E
76 FF FF FF 6F 06 DF 01 C3 BD 1E A7 3E 67 C5 BE 24 A9 DF 01 24 A8 FF 58
60 F7 76 7F 00 00 76 7F 00 4B B2 BD B8 42 AA BD C5 02 3E 67 B8 40 59 00
92 A1 EC 0D 56 03 01 A1 08 A9 10 40 5A A9 DA 82 C5 C2 3E 67 92 A1 9C FF
59 A9 ED FA B8 40 06 02 EC 03 1E A7 3E 67 B8 40 06 04 EC 03 1E A7 3E 67
77 00 6F 00 B8 42 B2 BD C5 02 8B A4 3E 67 B8 40 92 00 52 20 64 06 B8 42
C5 00 3E 67 9A 01 6F 0D 93 00 0A 00 56 03 01 A8 5C A9 08 A4 10 40 B8 42
B2 C4 C5 00 3E 67 9A 00 8B BE 00 06 6F 00 00 06 B8 42 A8 02 00 06 B8 42
A8 00 00 06
```

*Example 11−6. Sample Hex Convertor Out File for C28x 16-Bit Parallel Boot GP I/O (Continued)*

```
Length of second block in words
  |
  |       Address of the second block
  |       |
----- -----------
00 1A 00 3F 00 90 00 04 10 84 00 01 00 02 00 03 00 04 00 01 10 00 00 00
00 02 10 02 00 00 00 00 00 02 10 04 00 00 00 00 00 02 10 80 00 89 00 3F
00 02 10 82 00 89 00 3F 00 00 00 00
                              -----
                                |
                                Terminating header with length zero
```

The command file in Example 11−7 allows you to boot the .text and .cinit sections of test.out from a 16-bit-wide EPROM from the SCI-A 8-bit port. The map file test.map is also generated.

*Example 11−7. Sample Command File for Booting From 8-Bit SCI Boot*

```
/*----------------------------------------------------------------------*/
/* Hex convertor command file.                                          */
/*----------------------------------------------------------------------*/
test.out            /* Input COFF file */
-a                  /* Select ASCII format */
-map test.map       /* Specify the map file */
-o test_sci8.hex    /* Hex utility out file */
-sci8               /* Specify the SCI 8-bit boot format */

SECTIONS
{
   .text: paddr=BOOT
   .cinit: paddr=BOOT
}
```

The command file in Example 11−7 generates the out file in Example 11−8.

*Example 11−8. Sample Hex Convertor Out File for Booting From 8-Bit SCI Boot*

```
Key value
  |                     Reserved for future use
  |                       |                            Entry point
  |                       |                              |
----- ----------------------------------------------------- -----------
AA 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3F 00 05 00 90 00
-----
                                                                     |
                                       Length of first block in words

    Address of the first block
     |
-----------
3F 00 00 00 42 B8 00 9A 04 28 05 00 06 00 AD 28 88 10 69 FF 1F 56 16 56
1A 56 40 29 1F 76 00 00 02 29 1B 76 22 76 A9 28 90 00 A8 28 3F 00 01 09
1D 61 FF 76 90 00 04 29 0F 6F 00 9B A9 24 01 DF 04 6C 04 29 A8 24 01 DF
A6 1E A1 F7 86 24 A7 06 A1 81 01 09 A7 1E A9 24 03 63 5C FF 04 3B A9 59
00 77 00 77 01 DF 09 00 EA FF 1A 76 A9 28 FF FF A8 28 FF FF 01 09 0E 61
FF 76 FF FF 06 6F 01 DF BD C3 A7 1E 67 3E BE C5 A9 24 01 DF A8 24 58 FF
F7 60 7F 76 00 00 7F 76 4B 00 BD B2 42 B8 BD AA 02 C5 67 3E 40 B8 00 59
A1 92 0D EC 03 56 A1 01 A9 08 40 10 A9 5A 82 DA C2 C5 67 3E A1 92 FF 9C
A9 59 FA ED 40 B8 02 06 03 EC A7 1E 67 3E 40 B8 04 06 03 EC A7 1E 67 3E
00 77 00 6F 42 B8 BD B2 02 C5 A4 8B 67 3E 40 B8 00 92 20 52 06 64 42 B8
00 C5 67 3E 01 9A 0D 6F 00 93 00 0A 03 56 A8 01 A9 5C A4 08 40 10 42 B8
C4 B2 00 C5 67 3E 00 9A BE 8B 06 00 00 6F 06 00 42 B8 02 A8 06 00 42 B8
00 A8 06 00
Length of second block in words
  |
  |       Address of the second block
  |        |
----- -----------
1A 00 3F 00 90 00 04 00 84 10 01 00 02 00 03 00 04 00 01 00 00 10 00 00
02 00 02 10 00 00 00 00 02 00 04 10 00 00 00 00 02 00 80 10 89 00 3F 00
02 00 82 10 89 00 3F 00 00 00 00 00
                              -----
                         |
Terminating header with length zero
```

## 11.10 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 11-22) 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.

## 11.11 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 11−5 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 11−5. Options for Specifying Hex-Conversion Formats*

| Option | Format | Address Bits | Default Width |
|---|---|---|---|
| −a | ASCII-Hex | 16 | 8 |
| −i | Intel | 32 | 8 |
| −m | Motorola-S | 24 | 8 |
| −t | TI-Tagged | 16 | 16 |
| −x | Tektronix | 32 | 8 |

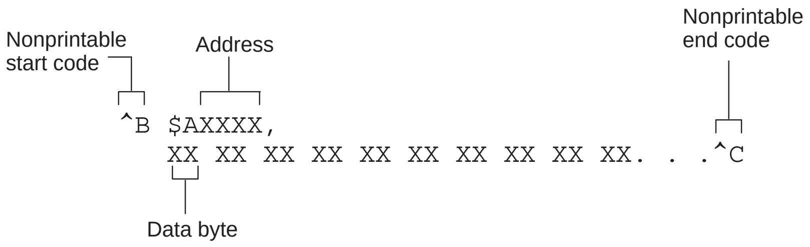
**Address bits** determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

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.

### 11.11.1 ASCII-Hex Object Format (−a Option)

The ASCII-Hex object format supports 16-bit addresses. The format consists of a byte stream with bytes separated by spaces. Figure 11−7 illustrates the ASCII-Hex format.

*Figure 11−7.ASCII-Hex Object Format*



```
^B $AXXXX,
   XX XX XX XX XX XX XX XX XX XX. . .^C
```

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.

### 11.11.2 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 |

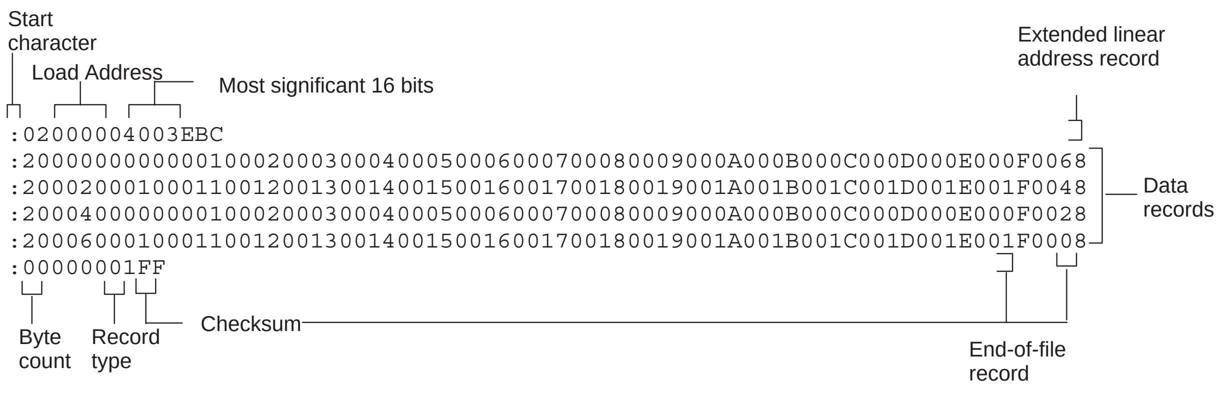
Record type *00*, the data record, begins with a colon ( : ), which is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 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 11−8 illustrates the Intel hexadecimal object format.

*Figure 11−8. Intel MCS86 Hexadecimal Object Format*



```
:02000004003EBC
:2000000000000100020003000400050006000700080009000A000B000C000D000E000F0068
:2000200010001100120013001400150016001700180019001A001B001C001D001E001F0048
:2000400000000100020003000400050006000700080009000A000B000C000D000E000F0028
:2000600010001100120013001400150016001700180019001A001B001C001D001E001F0008
:00000001FF
```

### 11.11.3 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:

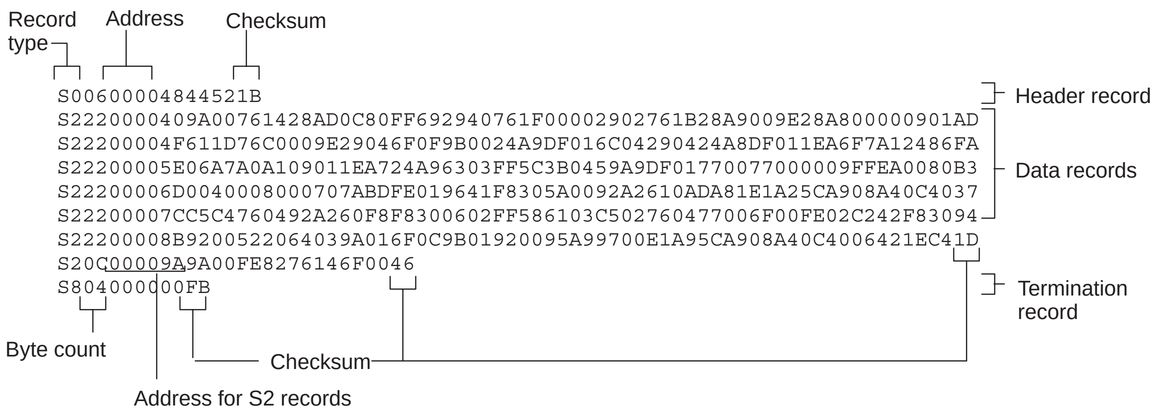
| Record Type | Description |
|---|---|
| S0 | Header record |
| S2 | Code/data record |
| S8 | Termination record |

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 11−9 illustrates the Motorola-S object format.

*Figure 11−9. Motorola-S Object Format*



```
S00600004844521B
S2220000409A00761428AD0C80FF692940761F00002902761B28A9009E28A800000901AD
S22200004F611D76C0009E29046F0F9B0024A9DF016C04290424A8DF011EA6F7A12486FA
S22200005E06A7A0A109011EA724A96303FF5C3B0459A9DF01770077000009FFEA0080B3
S22200006D0040008000707ABDFE019641F8305A0092A2610ADA81E1A25CA908A40C4037
S22200007CC5C4760492A260F8F8300602FF586103C502760477006F00FE02C242F83094
S22200008B9200522064039A016F0C9B01920095A99700E1A95CA908A40C4006421EC41D
S20C00009A9A00FE8276146F0046
S804000000FB
```

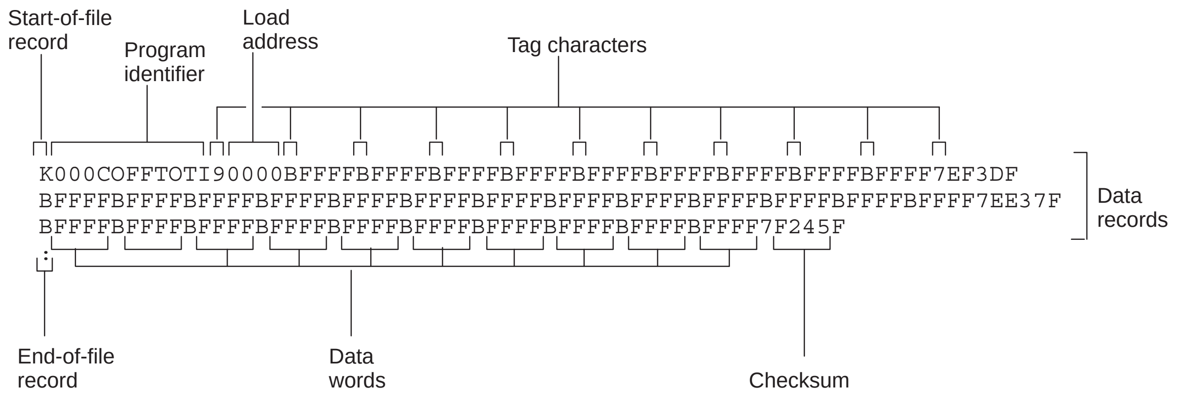
### 11.11.4 TI-Tagged SDSMAC Object Format (−t Option)

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 11−10 illustrates the tag characters and fields in TI-Tagged object format.

*Figure 11−10. TI-Tagged Object Format*



```
K000COFFTOTI90000BFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFF7EF3DF
BFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFF7EE37F
BFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFF7F245F
:
```

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 ( : ).

### 11.11.5 Extended Tektronix Object Format (−x Option)

The Tektronix object format supports 32-bit addresses and has two types of records:

| | |
|---|---|
| **Data records** | contain the header field, the load address, and the object code. |
| **Termination records** | signify the end of a module. |

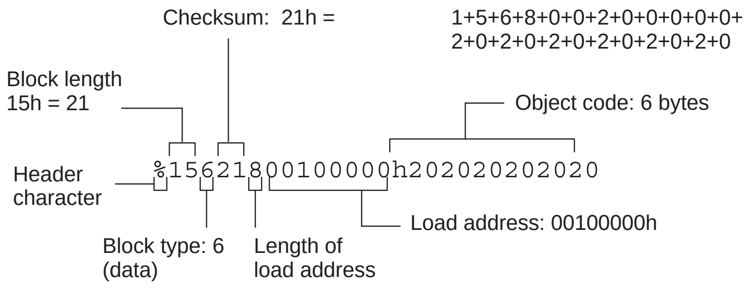
The header field in the data record contains the following information:

| 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<br>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 11−11 illustrates the Tektronix object format.

*Figure 11−11.Extended Tektronix Object Format*



## 11.12 Hex-Conversion Utility Error Messages

**section mapped to reserved memory**

*Description* A section is mapped into a memory area that is designated as reserved in the processor memory map.

*Action* Correct section or boot-loader address. For valid memory locations, refer to the *TMS320C28x CPU and Instruction Set Reference Guide*.

**sections overlapping**

*Description* Two or more COFF section load addresses overlap, or a boot table address overlaps another section.

*Action* This problem may be caused by an incorrect translation from load address to hexadecimal output-file address that is performed by the hex-conversion utility when memory width is less than data width. See section 11.3, *Understanding Memory Widths*, on page 11-8 and section 11.10, *Controlling the ROM Device Address*, on page 11-38.

**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.

# Common Object File Format

The TMS320C28x™ 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 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-2**
**A.2 File Header Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-4**
**A.3 Optional File Header Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-5**
**A.4 Section Header Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-6**
**A.5 Structuring Relocation Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . A-9**
**A.6 Symbol Table Structure and Content . . . . . . . . . . . . . . . . . . . . . . . . . A-11**

## 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
- ❏ 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*

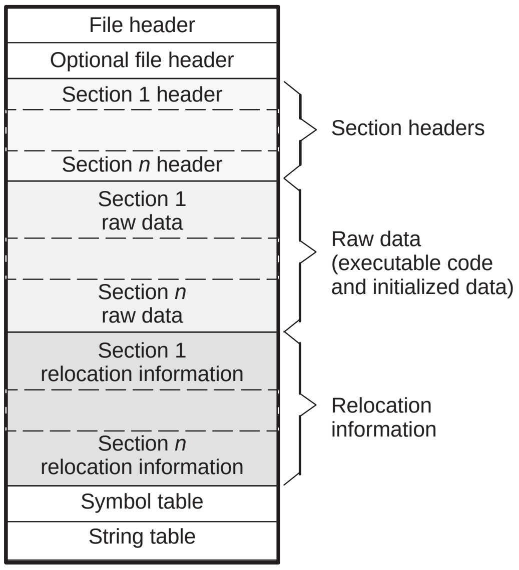
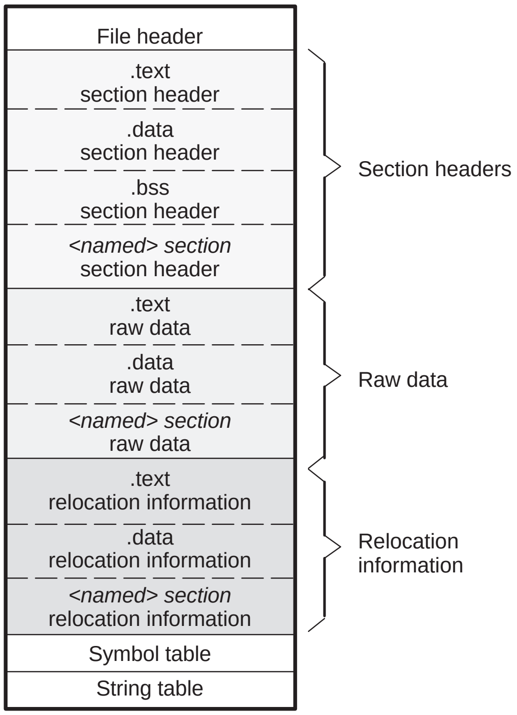


Figure A−2 shows a typical example of a COFF object file that contains the three default sections, .text, .data, and .bss, and a named section (referred to as <named>). By default, the tools place sections into the object file in the following order: .text, .data, initialized named sections, .bss, and 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 TMS320C28x 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 | Long | Time and date stamp; indicates when the file was created |
| 8−11 | Long | File pointer; contains the symbol table's starting address |
| 12−15 | Long | 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 (009dh) indicates the file can be executed in a TMS320C28x 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_LSYMS | 0008h | Local symbols were stripped from the file. |
| F_LITTLE | 0100h | The target is a little-endian device. |
| F_SYMMERGE | 1000h | Duplicate symbols were removed. |

## A.3 Optional File Header Format

The linker creates the optional file header and uses it to perform relocation at 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 | Long | Size (in bytes) of executable code |
| 8−11 | Long | Size (in bytes) of initialized data |
| 12−15 | Long | Size (in bytes) of uninitialized data |
| 16−19 | Long | Entry point |
| 20−23 | Long | Beginning address of executable code |
| 24−27 | Long | 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 | Long | Section’s physical address |
| 12−15 | Long | Section’s virtual address |
| 16−19 | Long | Section’s size in words |
| 20−23 | Long | File pointer to raw data |
| 24−27 | Long | File pointer to relocation entries |
| 28−31 | Long | Reserved |
| 32−35 | Unsigned long | Number of relocation entries |
| 36−39 | Unsigned long | Reserved |
| 40−43 | Unsigned long | 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)*

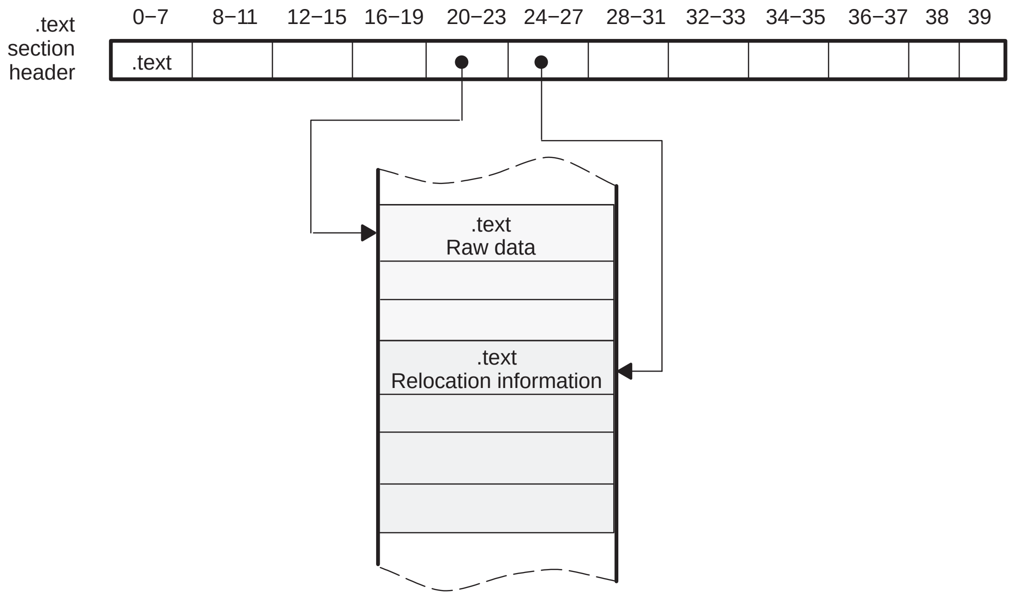
| Mnemonic | Flag | Description |
|---|---|---|
| STYP_REG | 00000000h | Regular section (allocated, relocated, loaded) |
| STYP_DSECT | 00000001h | Dummy section (relocated, not allocated, not loaded) |
| STYP_NOLOAD | 00000002h | Noload section (allocated, relocated, not loaded) |
| STYP_GROUP | 00000004h | 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 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 or relocation information. They occupy no actual space in the object file. Therefore, the number of relocation 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 | Long | Virtual address of the reference |
| 4−5 | Unsigned long | Symbol-table index (0−65535) |
| 6−7 | Unsigned short | Additional byte used for extended addressing calculations |
| 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:

```
1                          .global   X
2   00000000  FFEF!           b      X, unc
    0000001   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 contains 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), this is called an *internal relocation.* In this case, the relocation amount is simply the amount by which the current section is being relocated.

- The **relocation type** specifies the size of the field to be patched and describes how 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_C28PCR16. Table A−7 lists the relocation types.

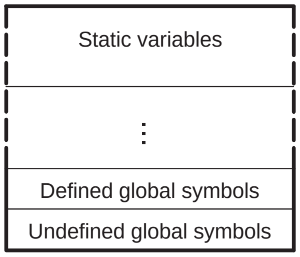
*Table A−7. Relocation Types (Bytes 8 and 9)*

| 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_C28PCR16 | 062h | PC relative 16-bit |
| R_C28PCR8 | 063h | PC relative 8-bit |
| R_C28PTR | 064h | 22-bit pointer |
| R_C28HI16 | 065h | High 16 bits of address data |
| R_C28LOPTR | 066h | Pointer to low 64K |
| R_C28NWORD | 067h | 16-bit negated relocation |
| R_C28NBYTE | 068h | 8-bit negated relocation |
| R_C28HIBYTE | 069h | High 8 bits of a 16-bit data |
| R_C28RELS13 | 06ah | Signed 13-bit value relocated as 16-bit value. |

## A.6 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−4.

*Figure A−4. 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

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−8. Each symbol may also have an 18-byte auxiliary entry; the special symbols listed in Table A−9, page A-12, 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−8. Symbol-Table Entry Contents*

| Byte Number | Type | Description |
|---|---|---|
| 0−7 | Char | This field contains one of the following:<br>❑ An 8-character symbol name, padded with nulls<br>❑ A pointer into the string table if the symbol name is longer than eight characters |
| 8−11 | Long | Symbol value; storage class dependent |
| 12−13 | Short | Section number of the symbol |
| 14−15 | Unsigned short | Reserved |
| 16 | Char | Storage class of the symbol |
| 17 | Char | Number of auxiliary entries (always 0 or 1) |

### A.6.1 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−9 lists these symbols.

*Table A−9. 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 |
| 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 |

## A.6.2 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.6.3 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−5 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−5. 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.6.4 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−10 lists valid storage classes.

*Table A−10. 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 | Reserved | C_ENTAG | 15 | Reserved |
| C_EXT | 2 | External definition | C_MOE | 16 | Reserved |
| C_STAT | 3 | Static | C_REGPARM | 17 | Reserved |
| C_REG | 4 | Reserved | C_FIELD | 18 | Reserved |
| 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 | Reserved | C_BLOCK | 100 | Reserved |
| C_ARG | 9 | Reserved | C_FCN | 101 | Reserved |
| C_STRTAG | 10 | Reserved | C_EOS | 102 | Reserved |
| C_MOU | 11 | Reserved | C_FILE | 103 | Reserved |
| C_UNTAG | 12 | Reserved | C_LINE | 104 | Used only by utility programs |
| C_TPDEF | 13 | Reserved | | | |

The .text, .data, and .bss symbols are restricted to the C_STAT storage class.

### A.6.5 Symbol Values

Bytes 8−11 of a symbol-table entry indicate a symbol's value. The C_EXT, C_STAT, and C_LABEL storage classes hold relocatable addresses. 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.6.6 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−11 lists these numbers and the sections they indicate.

*Table A−11. Section Numbers*

| Mnemonic | Section Number | Description |
|---|---|---|
| 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 −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.6.7 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). Table A−12 illustrates the format of auxiliary table entries for .text, .data, and .bss sections.

*Table A−12. Section Format for Auxiliary Table Entries*

| Byte Number | Type | Description |
|---|---|---|
| 0−3 | Long | 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) |

Appendix B

# Symbolic Debugging Directives

The assembler supports several directives that the TMS320C28x C/C++ compiler uses for symbolic debugging. These directives differ for the two debugging formats, DWARF and COFF.

These directives are not meant for use by assembly-language programmers. They require arguments that can be difficult to calculate manually, and their usage must conform to a predetermined agreement between the compiler, the assembler, and the debugger. This appendix documents these directives for informational purposes only.

**Topic** **Page**

**B.1 DWARF Debugging Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-2**
**B.2 COFF Debugging Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-3**
**B.3 Debug Directive Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-4**

## B.1 DWARF Debugging Format

A subset of the DWARF symbolic debugging directives are always listed in the assembly language file that the compiler creates for program analysis purposes. To list the complete set used for full symbolic debug, invoke the compiler with the −g option, as shown below:

```
cl2000 −v28 −g −k input_file
```

The −k option instructs the compiler to retain the generated assembly file.

To disable the generation of all symbolic debug directives, invoke the compiler with the −symdebug:none option:

```
cl2000 −v28 --symdebug:none −k input_file
```

The DWARF debugging format consists of the following directives:

- The **.dwtag** and **.dwendtag** directives define a Debug Information Entry (DIE) in the .debug_info section.
- The **.dwattr** directive adds an attribute to an existing DIE.
- The **.dwpsn** directive identifies the source position of a C/C++ statement.
- The **.dwcie** and **.dwendentry** directives define a Common Information Entry (CIE) in the .debug_frame section.
- The **.dwfde** and **.dwendentry** directives define a Frame Description Entry (FDE) in the .debug_frame section.
- The **.dwcfa** directive defines a call frame instruction for a CIE or FDE.

## B.2 COFF Debugging Format

COFF symbolic debug is now obsolete. These directives are supported for backwards-compatibility only. The decision to switch to DWARF as the symbolic debug format was made to overcome many limitations of COFF symbolic debug, including the absence of C++ support.

The COFF debugging format consists of the following directives:

- ❑ The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the variable or function.
- ❑ The **.stag**, **.etag**, and **.utag** directives define structures, enumerations, and unions, respectively. The **.membe**r directive specifies a member of a structure, enumeration, or union. The **.eos** directive ends a structure, enumeration, or union definition.
- ❑ The **.func** and **.endfunc** directives specify the beginning and ending lines of a C/C++ function.
- ❑ The **.block** and **.endblock** directives specify the bounds of C/C++ blocks.
- ❑ The **.file** directive defines a symbol in the symbol table that identifies the current source filename.
- ❑ The **.line** directive identifies the line number of a C/C++ source statement.

## B.3 Debug Directive Syntax

Table B−1 is an alphabetical listing of the symbolic debugging directives. For information on the C/C++ compiler, refer to the *TMS320C28x Optimizing C/C++ Compiler User's Guide.*

*Table B−1. Symbolic Debugging Directives*

| Label | Directive | Arguments |
|---|---|---|
| | **.block** | [*beginning line number*] |
| | **.dwattr** | *DIE label*, *DIE attribute name*(*DIE attribute value*)[,*DIE attribute name*(*attribute value*) [, ...] |
| | **.dwcfa** | *call frame instruction opcode* [, *operand* [, *operand*]] |
| *CIE label* | **.dwcie** | *version*, *return address register* |
| | **.dwendentry** | |
| | **.dwendtag** | |
| | **.dwfde** | *CIE label* |
| | **.dwpsn** | "*filename*", *line number*, *column number* |
| *DIE label* | **.dwtag** | *DIE tag name*, *DIE attribute name*(*DIE attribute value*)[,*DIE attribute name*(*attribute value*) [, ...] |
| | **.endblock** | [*ending line number*] |
| | **.endfunc** | [*ending line number* [, *register mask* [, *frame size* ]]] |
| | **.eos** | |
| | **.etag** | *name* [, *size*] |
| | **.file** | "*filename*" |
| | **.func** | [*beginning line number*] |
| | **.line** | *line number* [, *address*] |
| | **.member** | *name*, *value* [, *type*, *storage class*, *size*, *tag*, *dims*] |
| | **.stag** | *name* [, *size*] |
| | **.sym** | *name*, *value* [, *type*, *storage class*, *size*, *tag*, *dims*] |
| | **.utag** | *name* [, *size*] |

Appendix C

# XML Link Information File Description

The linker supports the generation of an XML link information file via the −−xml_link_info *file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker-generated map file.

As the linker evolves, the XML link information file may be extended to include additional information that could be useful for static analysis of linker results.

This appendix enumerates all of the elements that are generated by the linker into the XML link information file.

**Topic** **Page**

**C.1 XML Information File Element Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-2**
**C.2 Document Elements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-3**

## C.1 XML Information File Element Types

These element types will be generated by the linker:

- **Container elements** represent an object that contains other elements that describe the object. Container elements have an id attribute that makes them accessible from other elements.
- **String elements** contain a string representation of their value.
- **Constant elements** contain a 32-bit unsigned long representation of their value (with a 0x prefix).
- **Reference elements** are empty elements that contain an idref attribute that specifies a link to another container element.

In section C.2, the element type is specified for each element in parentheses following the element description. For instance, the <link_time> element lists the time of the link execution (string).

## C.2 Document Elements

The root element, or the document element, is **<link_info>**. All other elements contained in the XML link information file are children of the <link_info> element. The following sections describe the elements that an XML information file can contain.

### C.2.1 Header Elements

The first elements in the XML link information file provide general information about the linker and the link session:

- The **<banner>** element lists the name of the executable and the version information (string).
- The **<copyright>**element lists the TI copyright information (string).
- The **<link_time>** element lists the time of the link execution (string).
- The **<link_timestamp>** is a timestamp representation of the link time (unsigned 32-bit int)
- The **<output_file>** element lists the name of the linked output file generated (string).
- The **<entry_point>** element specifies the program entry point, as determined by the linker (container) with two entries:
  - The **<name>** is the entry point symbol name, if any (string).
  - The **<address>** is the entry point address (constant).

*Example C−1. Header Element for the hi.out Output File*

```
<banner>TMS320Cxx COFF Linker       Version x.xx (Jan  6 2003)</banner>
<copyright>Copyright (c) 1996−2003 Texas Instruments Incorporated</copyright>
<link_time>Mon Jan  6 15:38:18 2003</link_time>
<output_file>hi.out</output_file>
<entry_point>
   <name>_c_int00</name>
   <address>0xaf80</address>
</entry_point>
```

### C.2.2 Input File List

The next section of the XML link information file is the input file list, which is delimited with a **<input_file_list>** container element. The <input_file_list> can contain any number of <input_file> elements.

Each **<input_file>** instance specifies the input file involved in the link. Each <input_file> has an id attribute that can be referenced by other elements, such as an <object_component>. An <input_file> is a container element enclosing the following elements:

❑ The **<path>** element names a directory path, if applicable (string).

❑ The **<kind>** element specifies a file type, either archive or object (string).

❑ The **<file>** element specifies an archive name or filename (string).

❑ The **<name>** element specifies an object file name, or archive member name (string).

*Example C−2. Input File List for the hi.out Output File*

```
<input_file_list>
   <input_file id=”fl−1”>
      <kind>object</kind>
      <file>hi.obj</file>
      <name>hi.obj</name>
   </input_file>
   <input_file id=”fl−2”>
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>boot.obj</name>
   </input_file>
   <input_file id=”fl−3”>
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>exit.obj</name>
   </input_file>
   <input_file id=”fl−4”>
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>printf.obj</name>
   </input_file>
    ...
</input_file_list>
```

### C.2.3 Object Component List

The next section of the XML link information file contains a specification of all of the object components that are involved in the link. An example of an object component is an input section. In general, an object component is the smallest piece of object that can be manipulated by the linker.

The **<object_component_list>** is a container element enclosing any number of <object_component> elements.

Each **<object_component>** specifies a single object component. Each <object_component> has an id attribute so that it can be referenced directly from other elements, such as a <logical_group>. An <object_component> is a container element enclosing the following elements:

- ❑ The **<name>** element names the object component (string).
- ❑ The **<load_address>** element specifies the load-time address of the object component (constant).
- ❑ The **<run_address>** element specifies the run-time address of the object component (constant).
- ❑ The **<size>** element specifies the size of the object component (constant).
- ❑ The **<input_file_ref>** element specifies the source file where the object component originated (reference).

*Example C−3. Object Component List for the fl−4 Input File*

```
<object_component id=”oc-20”>
   <name>.text</name>
   <load_address>0xac00</load_address>
   <run_address>0xac00</run_address>
   <size>0xc0</size>
   <input_file_ref idref=”fl-4”/>
</object_component>
<object_component id=”oc-21”>
   <name>.data</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref=”fl-4”/>
</object_component>
<object_component id=”oc-22”>
   <name>.bss</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref=”fl-4”/>
</object_component>
```

### C.2.4 Logical Group List

The **<logical_group_list>** section of the XML link information file is similar to the output section listing in a linker generated map file. However, the XML link information file contains a specification of GROUP and UNION output sections, which are not represented in a map file. There are three kinds of list items that can occur in a <logical_group_list>:

- The **<logical_group>** is the specification of a section or GROUP that contains a list of object components or logical group members. Each <logical_group> element is given an id so that it may be referenced from other elements. Each <logical_group> is a container element enclosing the following elements:
  - The **<name>** element names the logical group (string).
  - The **<load_address>** element specifies the load-time address of the logical group (constant).
  - The **<run_address>** element specifies the run-time address of the logical group (constant).
  - The **<size>** element specifies the size of the logical group (constant).
  - The **<contents>** element lists elements contained in this logical group (container). These elements refer to each of the member objects contained in this logical group:
    - The **<object_component_ref>** is an object component that is contained in this logical group (reference).
    - The **<logical_group_ref>** is a logical group that is contained in this logical group (reference).
- The **<overlay>** is a special kind of logical group that represents a UNION, or a set of objects that share the same memory space (container). Each <overlay> element is given an id so that it may be referenced from other elements (like from an <allocated_space> element in the placement map). Each <overlay> contains the following elements:
  - The **<name>** element names the overlay (string).
  - The **<run_address>** element specifies the run−time address of overlay (constant).
  - The **<size>** element specifies the size of logical group (constant).

  - The **<contents>** container element lists elements contained in this overlay. These elements refer to each of the member objects contained in this logical group:
    - The **<object_component_ref>** is an object component that is contained in this overlay (reference).
    - The **<logical_group_ref>** is a logical group that is contained in this overlay (reference).

- The **<split_section>** is another special kind of logical group which represents a collection of logical groups that is split among multiple memory areas. Each <split_section> element is given an id so that it may be referenced from other elements. The id consists of the following elements.
  - The **<name>** element names the split section (string).
  - The **<contents>** element lists elements contained in this split section (container). The <logical_group_ref> elements refer to each of the member objects contained in this split section, and each element referenced is a logical group that is contained in this split section (reference).

*Example C−4. Logical Group List for the fl−4 Input File*

```
<logical_group_list>
   ...
      <logical_group id="lg-7">
      <name>.text</name>
      <load_address>0x20</load_address>
      <run_address>0x20</run_address>
      <size>0xb240</size>
      <contents>
         <object_component_ref idref="oc-34"/>
         <object_component_ref idref="oc-108"/>
         <object_component_ref idref="oc-e2"/>
      ...
      </contents>
   </logical_group>
   ...
   <overlay id="lg-b">
      <name>UNION_1</name>
      <run_address>0xb600</run_address>
      <size>0xc0</size>
      <contents>
         <object_component_ref idref="oc-45"/>
         <logical_group_ref idref="lg-8"/>
      </contents>
   </overlay>
    ...
   <split_section id="lg-12">
      <name>.task_scn</name>
      <size>0x120</size>
      <contents>
         <logical_group_ref idref="lg-10"/>
         <logical_group_ref idref="lg-11"/>
      </contents>
   ...
</logical_group_list>
```

## C.2.5 Placement Map

The **<placement_map>** element describes the memory placement details of all named memory areas in the application, including unused spaces between logical groups that have been placed in a particular memory area.

- The **<memory_area>** is a description of the placement details within a named memory area (container). The description consists of these items:
  - The **<name>** names the memory area (string).
  - The **<page_id>** gives the id of the memory page in which this memory area is defined (constant).
  - The **<origin>** specifies the beginning address of the memory area (constant).
  - The **<length>** specifies the length of the memory area (constant).
  - The **<used_space>** specifies the amount of allocated space in this area (constant).
  - The **<unused_space>** specifies the amount of available space in this area (constant).
  - The **<attributes>** lists the RWXI attributes that are associated with this area, if any (string).
  - The **<fill_value>** specifies the fill value that is to be placed in unused space, if the fill directive is specified with the memory area (constant).
  - The **<usage_details>** lists details of each allocated or available fragment in this memory area. If the fragment is allocated to a logical group, then a <logical_group_ref> element is provided to facilitate access to the details of that logical group. All fragment specifications include <start_address> and <size> elements.
    - The **<allocated_space>** element provides details of an allocated fragment within this memory area (container):

      The **<start_address>** specifies the address of the fragment (constant).

      The **<size>** specifies the size of the fragment (constant).

      The **<logical_group_ref>** provides a reference to the logical group that is allocated to this fragment (reference).
    - The **<available_space>** element provides details of an available fragment within this memory area (container):

      The **<start_address>** specifies the address of the fragment (constant).

      The **<size>** specifies the size of the fragment (constant).

*Example C−5. Placement Map for the fl−4 Input File*

```
<placement_map>
   <memory_area>
      <name>PMEM</name>
      <page_id>0x0</page_id>
      <origin>0x20</origin>
      <length>0x100000</length>
      <used_space>0xb240</used_space>
      <unused_space>0xf4dc0</unused_space>
      <attributes>RWXI</attributes>
      <usage_details>
         <allocated_space>
            <start_address>0x20</start_address>
            <size>0xb240</size>
            <logical_group_ref idref=”lg-7”/>
         </allocated_space>
         <available_space>
            <start_address>0xb260</start_address>
            <size>0xf4dc0</size>
         </available_space>
      </usage_details>
   </memory_area>
   ...
</placement_map>
```

### C.2.6 Symbol Table

The **<symbol_table>** contains a list of all of the global symbols that are included in the link. The list provides information about a symbol's name and value. In the future, the symbol_table list may provide type information, the object component in which the symbol is defined, storage class, etc.

The **<symbol>** is a container element that specifies the name and value of a symbol with these elements:

- The **<name>** element specifies the symbol name (string)
- The **<value>** element specifies the symbol value (constant)

*Example C−6. Symbol Table for the fl−4 Input File*

```
<symbol_table>
   <symbol>
      <name>_c_int00</name>
      <value>0xaf80</value>
   </symbol>
   <symbol>
      <name>_main</name>
      <value>0xb1e0</value>
   </symbol>
   <symbol>
      <name>_printf</name>
      <value>0xac00</value>
   </symbol>
   ...
</symbol_table>
```

Appendix D

# Glossary

## A

**absolute address:** An address that is permanently assigned to a TMS320C28x™ 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 TMS320C28x 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 TMS320C28x, 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 TMS320C28x 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 TMS320C28x 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 TMS320C28x.

**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 TMS320C28x 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 TMS320C28x 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 TMS320C28x 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 *typ*e name that can be assigned to a structure, union, or enumeration.

**target memory:** Physical memory in a TMS320C28x 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

**A**

−a, name utility option 10-16
a archiver command 6-4
−a option
  hex conversion utility 11-5 , 11-40
  linker 7-6
A_DIR environment variable 3-8 , 7-11 , 7-12
−aa option, assembler option 3-4
absolute lister
  creating the absolute listing file 8-2
  described 1-4
  development flow 8-2
  example 8-5 to 8-10
  invoking 8-3
  options 8-3
absolute listing
  −aa assembler option 3-4
  producing 8-2
absolute output module 7-6
−ac option, assembler 3-4
−ad option, assembler 3-4
address specification, page method 7-35
−ahc assembler option 3-4
−ahi assembler option 3-5
−al option, assembler 3-5
.align directive
  compatibility with C1x/C2x/C2xx/C5x 4-7
  described 4-13
  reference 4-22
alignment 4-13 to 4-14 , 7-34
  defined D-1
  reference 4-22
allocation 2-2 , 7-32 to 7-38
  alignment 4-22 , 7-34
  allocating output sections 7-28
  binding 7-33
  blocking 7-34
  default algorithm 7-51 to 7-52
  defined D-1
  GROUP 7-45
  memory, default 2-12 to 2-13 , 7-33 to 7-34
  UNION 7-43
  variables 4-27
alternate directories 3-7 to 3-9 , 7-11
  naming with −i option 3-7
  naming with A_DIR 3-8
−ar linker option 7-7
ar2000 command 6-4
archive libraries
  back referencing 7-17
  defined D-1
  described 6-1
  exhaustively reading 7-17
  macro library 4-59
  object libraries 7-22 to 7-23
  specifying to linker 7-10
  types of files 6-2
archive library, allocating individual members 7-37
archiver 1-3 , 6-1 to 6-8
  commands
    *@ 6-4*
    *a 6-4*
    *d 6-4*
    *r 6-4*
    *t 6-4*
    *u 6-5*
    *x 6-4*
  defined D-1
  examples 6-6
  in the development flow 6-3
  invoking 6-4
  options
    *−q 6-5*
    *−s 6-5*
    *−v 6-5*

arithmetic operators 3-25 to 3-26

−as option, assembler 3-5

ASCII-Hex object format 11-1 , 11-40

.asg directive
- described 4-19
- listing control 4-15 , 4-37
- reference 4-23
- use in macros 5-6

asm2000 command 3-4

.asmfunc directive 4-21 , 4-25

assembler 1-3 , 3-1 to 3-46
- built-in math functions 3-29
- character strings 3-17
- constants 3-13 to 3-16
- cross-reference listings 3-5 , 3-36
- defined D-1
- expressions 3-25 to 3-28
- handling COFF sections 2-4 to 2-10
- in the development flow 3-3
- invoking 3-4
- macros 5-1 to 5-24
- options
  - *−aa 3-4*
  - *−ac 3-4*
  - *−ad 3-4*
  - *−ahc 3-4*
  - *−ahi 3-5*
  - *−al 3-5*
  - *−as 3-5*
  - *−au 3-5*
  - *−ax 3-5*
  - *−d 3-21*
  - *−g 3-5*
  - *−i 3-5 , 3-7*
  - *−l 3-31*
  - *−m20 3-5*
  - *−m27 3-5*
  - *−mf 3-6*
  - *−mg 3-6 , 3-39*
  - *−mw 3-6*
  - *−q 3-6*
  - *−x 3-36*
- output listing 3-33 , 4-15 to 4-16
  - *directive listing 4-15*
  - *enabling 4-15*
  - *false conditional block listing 4-15*
  - *list options 4-15 to 4-16*
  - *page eject 4-16*
  - *page length 4-15*
  - *page width 4-16*
  - *suppressing 4-15*
  - *tab size 4-16*
  - *title 4-16*
- relocation 2-14 to 2-16
  - *at run time 2-16*
  - *linker capabilties 7-6 to 7-7*
- sections directives 2-4 to 2-10
- source listings 3-31 to 3-33
- source statement format 3-10 to 3-12
- symbols 3-18 to 3-24
- TMS320C28x modes 3-41

assembler directives 4-1 to 4-84
- align section program counter 4-13
- default directive 2-4
- defining assembly-time symbols 4-19 to 4-20
  - *.asg 4-19*
  - *.endstruct 4-19*
  - *.eval 4-19*
  - *.label 4-19*
  - *.set 4-19*
  - *.struct 4-19*
  - *.tag 4-19*
- defining sections 4-8 to 4-9
  - *.bss 2-4 , 4-8*
  - *.data 2-4 , 4-8*
  - *.sect 2-4 , 4-8*
  - *.text 2-4 , 4-8*
  - *.usect 2-4 , 4-8*
- enabling conditional assembly 4-18
  - *.break 4-18*
  - *.else 4-18*
  - *.elseif 4-18*
  - *.endif 4-18*
  - *.endloop 4-18*
  - *.if 4-18*
  - *.loop 4-18*
- example 2-8 to 2-10
- formatting the output listing 4-15 to 4-16
  - *.drlist 4-15*
  - *.drnolist 4-15*
  - *.fclist 4-15*
  - *.fcnolist 4-15*
  - *.length 4-15*
  - *.list 4-15*
  - *.mlist 4-15*
  - *.mnolist 4-15*
  - *.nolist 4-15*
  - *.option 4-15 to 4-16*
  - *.page 4-16*

*.sslist 4-16*
*.ssnolist 4-16*
*.tab 4-16*
*.title 4-16*
*.width 4-16*
initializing constants 4-10 to 4-12
*.byte 4-10*
*.char 4-10*
*.field 4-10*
*.float 4-11*
*.int 4-11*
*.long 4-11*
*.short 4-11*
*.string 4-11*
*.word 4-11*
*.xfloat 4-11*
*.xstring 4-11*
miscellaneous directives 4-21
*.asmfunc 4-21 , 4-25*
*.clink 4-21 , 4-32*
*.emsg 4-21*
*.end 4-21*
*.endasmfunc 4-21 , 4-25*
*.mmsg 4-21*
*.newblock 4-21*
*.wmsg 4-21*
overriding assembler syntax mode 4-20 , 4-31
referencing other files 4-17
*.copy 4-17*
*.def 4-17*
*.global 4-17*
*.include 4-17*
*.mlib 4-17*
*.ref 4-17*
summary table 4-2 to 4-6

assembly language development flow 1-2 , 3-3 , 6-3 , 7-3

assembly variables, C-type symbolic debugging 3-39

assembly-time constants
defined D-1
described 3-16
reference 4-70

assignment expressions 7-54 to 7-55

attributes 3-36 , 7-27

−au option, assembler 3-5

autoinitialization
at load time 7-8 , 7-84
at run time 7-8 , 7-83
defined D-1

auxiliary entries A-15 to A-16
defined D-2

−ax option, assembler 3-5

## B

−b linker option 7-8

.bes directive 4-10 , 4-71

big endian
defined D-2
ordering 11-13

binary integer constants 3-13

binding 7-33
defined D-2

blocking 4-27 , 7-34

−boot hex conversion utility option 11-5 , 11-29

boot loader 7-85

boot.obj module 7-82 , 7-85

boot-time copy table generated by linker 7-69 to 7-70

−bootorg hex conversion utility option 11-5 , 11-29

.break directive 4-18
listing control 4-15 , 4-37
reference 4-57
use in macros 5-14 to 5-15

.bss directive 2-4

.bss directive 4-8 , 4-27
compatibility with C1x/C2x/C2xx/C5x 4-7

.bss linker symbol 7-56

.bss section 4-8 , 4-27 , A-3
defined D-2
holes 7-63 to 7-64
initializing 7-64

built-in math functions 3-29

.byte directive
described 4-10
limiting listing with the .option directive 4-15 , 4-65
reference 4-30

## C

−c, name utility option 10-16

C code, linking 7-82 to 7-85
example 7-86 to 7-89

C debugging for assembly variables 3-39

C hardware stack 7-83
C memory pool 7-10 , 7-83
−c option, linker 7-8 , 7-56 , 7-83
C software stack 7-83
C system stack 7-15
C/C++ compiler 1-3
  defined D-2
  linking conventions 7-8
  special symbols A-12
  storage classes A-14
  symbol table entries A-12
  symbolic debugging A-1
  symbolic debugging directives B-1 to B-14
C_DIR environment variable 7-11 , 7-12
_c_int00 7-9 , 7-85
C27x object mode 3-41
.c28_amode directive 4-20 , 4-31
C28x object −− accept C27x syntax mode 3-43
C28x object −− accept C2xLP syntax mode 3-43
C28x object mode 3-42
C2xLP syntax mode 3-43
.char directive
  described 4-10
  reference 4-30
character constants 3-15
character strings 3-17
.clink directive 4-21 , 4-32
COFF 2-1 to 2-20 , 7-1 , A-1 to A-16
  auxiliary entries A-15 to A-16
  conversion to hexadecimal format 11-1 to 11-46

  default allocation 7-51 to 7-52
  file headers A-4
  file structure A-2 to A-3
  initialized sections 2-5 to 2-6
  loading a program 2-17
  object file example A-3
  optional file header A-5
  relocation 2-14 to 2-16 , A-9 to A-10
    *relocation type A-10*
    *run-time relocation 2-16*
    *symbol table index A-9*
    *virtual address A-9*
  section headers A-6 to A-8
  sections 2-2 to 2-3
    *allocation 2-2*
    *assembler 2-4 to 2-10*
    *initialized 2-5 to 2-6*
    *linker 2-11 to 2-13*
    *named 2-6 to 2-7 , 7-61*
    *special types 7-50*
    *uninitialized 2-4 to 2-5*
  special symbols A-12
  storage classes A-14
  string table A-13
  symbol table
    *assembler usage 2-19 to 2-20*
    *linker usage 2-18*
    *structure and content A-11 to A-16*
    *symbol values A-14*
  uninitialized sections 2-4 to 2-5
command files
  defined D-2
  hex conversion utility 11-6 to 11-7
  linker 7-4 , 7-19 to 7-21
    *constants in 7-21*
    *example 7-87*
    *reserved words 7-21*
comment field 3-12
comments
  defined D-2
  in a linker command file 7-19 , 7-20
  in assembly language source code 3-12
  in macros 5-17
  source statement format 3-12
  that extend past page width 4-53
common object file format
  *See also* COFF
  defined D-2
conditional blocks 5-14 to 5-15
  assembly directives 4-18
    *in macros 5-14 to 5-15*
    *maximum nesting levels 5-14*
conditional expressions 3-27
conditional linking 4-32
conditional processing
  assembly directives, reference 4-49
  blocks
    *listing reference 4-41*
    *reference 4-49*
  defined D-2
configured memory 7-52
  defined D-2
constants 3-13 to 3-16 , 3-21 to 3-22
  assembly-time 3-16 , 4-70
  binary integers 3-13

character 3-15
decimal integers 3-14
defined D-3
floating-point 4-45
hexadecimal integers 3-15
in command files 7-21
octal integers 3-14
symbolic 3-22
*$ 3-22*
*processor symbols 3-23*
*status registers 3-23*
symbols as 3-21

.copy directive 3-7 , 4-17 , 4-33

copy file
.copy directive, reference 4-33
−ahc assembler option 3-4

copy files 3-7

copy routine, general-purpose 7-72 to 7-75

COPY section 7-50

copy tables automatically generated by linker 7-68 to 7-69
contents 7-70 to 7-72
sections and symbols 7-76 to 7-77

−cr linker option 7-8 , 7-56 , 7-84

creating holes 7-61 to 7-63

cross-reference lister 1-4 , 9-1 to 9-6
creating the cross-reference listing 9-2
development flow 9-2
example 9-4
invoking 9-3
listings 3-5 , 3-36
*defined D-3*
*producing with the .option directive 4-15 to 4-16*
options
*−l 9-3*
*−q 9-3*
symbol attributes 9-5
xref2000 command 9-3

cross-reference listing
example 9-4
producing with the .option directive 4-65

## D

−d, name utility option 10-16

d archiver command 6-4

−d option, assembler 3-21

.data directive 2-4 , 4-8
reference 4-36

.data linker symbol 7-56

.data section 4-8 , A-3
defined D-3

decimal integer constants 3-14

.def directive 4-17
identifying external symbols 2-18
reference 4-46

default
allocation 7-51 to 7-52
fill value for holes 7-9
memory allocation 2-12 to 2-13
MEMORY configuration 7-51 to 7-52
MEMORY model 7-24
SECTIONS configuration 7-29 , 7-51 to 7-52

defining macros 5-3 to 5-4

development tools overview 1-2

directives
assembler
*See also assembler directives*
*absolute lister 8-8*
defined D-3
linker
*MEMORY 2-11*
*SECTIONS 2-11*

directory search algorithm
assembler 3-7 to 3-9
linker 7-11

.drlist directive 4-15
reference 4-37
use in macros 5-20

.drnolist directive 4-15
reference 4-37
use in macros 5-20

DSECT section 7-50

dummy section 7-50

## E

−e, hex conversion utility option 11-31

−e hex conversion utility option 11-5 , 11-29

−e option
absolute lister 8-3
linker 7-9

edata linker symbol 7-56

.else directive 4-18
reference 4-49

use in macros 5-14 to 5-15
.elseif directive 4-18
reference 4-49
use in macros 5-14 to 5-15
.emsg directive 4-21 , 5-17
listing control 4-15 , 4-37
reference 4-38
end assembly 4-40
.end directive 4-21 , 4-40
end linker symbol 7-56
.endasmfunc directive 4-21 , 4-25
.endif directive 4-18
reference 4-49
use in macros 5-14 to 5-15
.endloop directive 4-18
reference 4-57
use in macros 5-14 to 5-15
.endm directive 5-3
.endstruct directive 4-19 , 4-74
entry points
assigning values to 7-9
_c_int00 7-9 , 7-85
default value 7-9
defined D-3
for C code 7-85
for the linker 7-9
_main 7-9
environment variables
A_DIR 3-8 , 7-11
C_DIR 7-10 to 7-12
EPROM programmer 1-4
error messages
generating 4-21
hex conversion utility 11-45
producing in macros 5-17
etext linker symbol 7-56
.eval directive
described 4-19
listing control 4-15 , 4-37
reference 4-23
use in macros 5-7
−exclude hex conversion utility option 11-4 , 11-23
executable module, defined D-3
executable output 7-6
relocatable 7-7
expressions 3-25 to 3-28
absolute and relocatable 3-27 to 3-28
*examples 3-28*
arithmetic operators 3-25 to 3-26
conditional 3-27
conditional operators 3-27
defined D-3
left-to-right evaluation 3-25
linker 7-54 to 7-55
overflow 3-26
parentheses effect on evaluation 3-25
precedence of operators 3-25
relocatable symbols 3-27 to 3-28
underflow 3-26
well-defined 3-27
external symbols 2-18 , 3-27
defined D-3
reference 4-46

F

−f, name utility option 10-16
−f linker option 7-9
.fclist directive 4-15
listing control 4-15 , 4-37
reference 4-41
use in macros 5-19
.fcnolist directive 4-15
listing control 4-15 , 4-37
reference 4-41
use in macros 5-19
.field directive 4-10
compatibility with C1x/C2x/C2xx/C5x 4-7
reference 4-42
file
copy 3-4
header defined D-3
header structure A-4
include 3-5
filenames
as character strings 3-17
copy/include files 3-7
extensions, changing defaults 8-3
macros, in macro libraries 5-13
files ROMS specification 11-16
−fill hex conversion utility option 11-4
fill MEMORY specification 7-27
−fill option, hex conversion utility 11-27
fill ROMS specification 11-16
fill value 7-63 to 7-64
default 7-9

setting 7-9
filling holes 7-63 to 7-64
.float directive 4-11
compatibility with C1x/C2x/C2xx/C5x 4-7
reference 4-45
floating-point constants 4-45

G

−g, name utility option 10-16
−g object file display option 10-2
−g option
assembler 3-5
linker 7-9
.global directive 4-17
identifying external symbols 2-18
reference 4-46
global symbols 7-10
defined D-4
making static with −h option 7-10
overriding −h option 7-9
−gpio16 hex conversion utility option 11-5 , 11-29 , 11-31 , 11-32
−gpio8 hex conversion utility option 11-5 , 11-29 , 11-31 , 11-32
GROUP statement 7-45
defined D-4

H

−h, name utility option 10-16
−h linker option 7-10
hardware stack, C language 7-83
−heap linker option 7-10
.hstack section 7-83
.sysmem section 7-10 , 7-83
hex conversion utility 1-4 , 11-1 to 11-45
command files 11-6 to 11-7
*invoking 11-3 , 11-6*
*ROMS directive 11-6*
*SECTIONS directive 11-6*
configuring memory widths
*defining memory word width (memwidth) 11-4*
*specifying output width (romwidth) 11-4*
controlling the boot table
*GP I/O port, 16-bit mode 11-5 , 11-29*
*GP I/O port, 8-bit mode 11-5 , 11-29*
*identifying bootable sections −boot 11-5 , 11-29*
*SCI-A port, 8-bit mode 11-5 , 11-29*
*setting the entry point −e 11-5 , 11-29*
*setting the ROM address −bootorg 11-5 , 11-29*
*speicifying LOSPCP initial value 11-5 , 11-29*
*speicifying SPIBRR initial value 11-5 , 11-29*
*SPI-A port, 8-bit mode 11-5 , 11-29*
defined D-4
error messages 11-45
excluding a specified section 11-23
generating a map file 11-4
generating a quiet run 11-4
hex2000 command 11-3
ignore specified section 11-4
image mode
*defining the target memory 11-27*
*filling holes 11-4 , 11-27*
*invoking 11-4 , 11-26*
*reset address origin 11-4*
*resetting address origin 11-26*
in the development flow 11-2
invoking 11-3 to 11-7
*from the command line 11-3*
*in a command file 11-3*
memory width (memwidth) 11-9 to 11-10
*exceptions 11-9*
on-chip boot loader 11-28 to 11-37
options
*−a 11-40*
*−fill 11-27*
*−i 11-41*
*−image 11-26*
*−m 11-42*
*−map 11-19 to 11-20*
*−memwidth 11-9*
*−o 11-24*
*−q 11-6*
*−romwidth 11-11*
*summary table 11-4*
*−t 11-43*
*−x 11-44*
ordering memory words 11-13 to 11-14
output filenames 11-4 , 11-24
*default filenames 11-24*
*ROMS directive 11-6*
ROM width (romwidth) 11-10 to 11-12

ROMS directive 11-15 to 11-20
*creating a map file of 11-19 to 11-46*
*defining the target memory 11-27*
*example 11-18 to 11-20*
*parameters 11-15 to 11-16*
*specifying output filenames 11-6*
SECTIONS directive 11-21 to 11-22
*parameters 11-21 to 11-22*
target width 11-9
hex2000 command 11-3
hexadecimal integers 3-15
holes 7-61 to 7-64
creating 7-61 to 7-63
defined D-4
fill value 7-30 , 11-16 , 11-27
filling 7-63 to 7-64 , 11-27
in output sections 7-61 to 7-64
in uninitialized sections 7-64
setting default fill value 7-9
__HSTACK_SIZE 7-56

**I**

I MEMORY attribute 7-27
−i option
assembler 3-5 , 3-7
*examples by operating system 3-8*
*maximum number per invocation 3-7*
hex conversion utility 11-5 , 11-41
linker 7-11
.if directive 4-18
reference 4-49
use in macros 5-14 to 5-15
−image hex conversion utility option 11-4 , 11-26
.include directive 3-7 , 4-17 , 4-33
include files 3-5 , 3-7 , 4-33
incremental linking 7-80 to 7-81
defined D-4
initialized sections 2-5 to 2-6 , 7-61
.data section 2-6 , 4-36
defined D-4
.sect section 2-6 , 4-69
subsections 2-6 , 2-7
.text section 2-6 , 4-78
input
linker 7-3 , 7-22 to 7-23
sections 7-36 to 7-38
*defined D-4*

.int directive 4-11 , 4-51
Intel object format 11-1 , 11-41
invoking
absolute lister 8-3
archiver 6-4
assembler 3-4
cross-reference lister 9-3
hex conversion utility 11-3 to 11-7
linker 7-4
object file display utility 10-2

**K**

keywords
allocation parameters 7-32
load 2-16 , 7-32 , 7-39
run 2-16 , 7-32 , 7-39

**L**

−l, name utility option 10-16
−l option
assembler, source listing format 3-31
cross-reference lister 9-3
linker 7-10
label
local 4-64
using with .byte directive 4-30
using with .char directive 4-30
.label directive 4-19 , 4-52
label field 3-11
labels 3-18
case sensitivity, −ac assembler option 3-4
defined D-4
defined and referenced (cross-reference list) 3-36
in assembly language source 3-11
in macros 5-16
local 3-18 to 3-20
symbols used as 3-18
syntax 3-11
large memory model option 3-6
left-to-right evaluation (of expressions) 3-25
Legal Expressions 3-27 to 3-28
.length directive 4-15
listing control 4-15
reference 4-53
length MEMORY specification 7-27

length ROMS specification 11-16
library search, using alternate mechanism, −priority linker option 7-17
library search algorithm 7-10 to 7-12
library-build utility 1-4
line-number table, line-number entries, defined D-4

linker 1-3 , 7-1 to 7-89
- archive members, allocating 7-37
- assigning symbols 7-53
- assignment expressions 7-54 to 7-55
- C code 7-82 to 7-85
- COFF 7-1
- command files 7-4 , 7-19 to 7-21
  - *example 7-87*
- configured memory 7-52
- defined D-4
- example 7-86 to 7-89
- generated copy tables. *See* linker-generated copy tables
- GROUP statement 7-43 , 7-45
- handling COFF sections 2-11 to 2-13
- in the development flow 7-3
- input 7-3 , 7-19 to 7-21
- invoking 7-4
- keywords 7-21 , 7-39 to 7-42
- linking C code 7-8 , 7-82 to 7-85
- lnk2000 command 7-4
- loading a program 2-17
- MEMORY directive 2-11 , 7-24 to 7-28
- object libraries 7-22 to 7-23
- operators 7-55
- options
  - *−a 7-6*
  - *−ar 7-7*
  - *−c 7-8 , 7-83*
  - *−cr 7-8 , 7-84*
  - *−e 7-9*
  - *−f 7-9*
  - *−g 7-9*
  - *−h 7-10*
  - *−heap 7-10*
  - *−i 7-11*
  - *−l 7-10*
  - *−m 7-13 to 7-14*
  - *−o 7-14*
  - *−q 7-15*
  - *−r 7-7*
  - *−s 7-15*
  - *−stack 7-15*
  - *summary table 7-5*
  - *−u 7-16*
  - *−w 7-16*
  - *−x 7-17*
- output 7-3 , 7-14 , 7-86
- overview 7-2
- partial linking 7-80 to 7-81
- section run-time address 7-39 to 7-42
- sections 2-13
  - *output 7-51*
  - *special 7-50*
- SECTIONS directive 2-11 , 7-29 to 7-38
- symbols 2-18 to 2-20 , 7-56
- table() operator 7-69 , 7-70
- unconfigured memory, overlaying 7-50
- UNION statement 7-43 to 7-44 , 7-67

linker directives
- MEMORY 7-24 to 7-28
- SECTIONS 7-29 to 7-38

linker-generated copy tables 7-65 to 7-79
- automatic 7-68 to 7-69
- boot-loaded application process 7-65
  - *alternative approach 7-66*
- boot-time copy table 7-69 to 7-70
- contents 7-70 to 7-72
- general-purpose copy routine 7-72 to 7-75
- overlay management 7-77 to 7-79
- overlay management example 7-67
- sections and symbols 7-76 to 7-77
- splitting object components 7-77 to 7-79
- table() operator 7-69
  - *manage object components 7-70*

.list directive 4-15 , 4-54

lister
- absolute 8-1 to 8-10
- cross-reference 9-1 to 9-6

listing
- control 4-15 to 4-16
- cross-reference listing 4-15 , 4-65
- file 4-15 to 4-16
  - *creating with the −al option 3-5*
  - *defined D-5*
  - *format 3-31 to 3-33*
- page eject 4-16
- page size 4-15
- reset the .option directive 4-15 , 4-65
- turn off listing of certain directives with the .option directive 4-15 , 4-65
- turn off with the .option directive 4-15 , 4-65
- turn on with the .option directive 4-15 , 4-65

little endian
- defined  D-5
- ordering  11-13

lnk2000 command  7-4

load address of a section  7-39
- referring to with a label  7-40 to 7-42

load linker keyword  2-16 , 7-39

LOAD_START() linker operator  7-66

loader  7-85
- defined  D-5

loading a program  2-17

local labels  3-18 to 3-20

logical operators  3-25 to 3-26

.long directive  4-11
- compatibility with C1x/C2x/C2xx/C5x  4-7
- limiting listing with the .option directive  4-15 to 4-16 , 4-65
- reference  4-56

.loop directive  4-18
- reference  4-57
- use in macros  5-14 to 5-15

−lospcp hex conversion utility option  11-5 , 11-29

.lp_amode directive  4-20 , 4-31

**M**

−m option
- hex conversion utility  11-5 , 11-42
- linker  7-13 to 7-14

−m20 assembler option  3-5 , 3-41

−m27 assembler option  3-5 , 3-41

macro
- disabling macro expansion listing  4-65
- example  5-4
- .mlib directive  4-59
- .mlist directive  4-61

.macro directive  4-58 , 5-3 to 5-4
- summary table  5-23 to 5-24

macros  5-1 to 5-24
- conditional assembly  5-14 to 5-15
- defined
  - *macro  D-5*
  - *macro call  D-5*
  - *macro definition  D-5*
  - *macro expansion  D-5*
  - *macro library  D-5*
- defining a macro  5-3 to 5-4
- description  5-2
- directives summary  5-23 to 5-24
- disabling macro expansion listing  4-15
- formatting the output listing  5-19 to 5-20
- labels  5-16
- macro comments  5-4 , 5-17
- macro libraries  5-13 , 6-2
  - *defined  D-5*
- nested macros  5-21 to 5-22
- parameters  5-5 to 5-12
- producing messages  5-17 to 5-18
- recursive macros  5-21 to 5-22
- substitution symbols  5-5 to 5-12
- using a macro  5-2

magic number, defined  D-5

_main  7-9

malloc() function  7-10 , 7-83

map file  7-13 to 7-14 , 11-19 to 11-20
- defined  D-5
- example  7-88 , 11-19

−map hex conversion utility option  11-4 , 11-19

math functions built-in for assembler  3-29

memory
- allocation  7-51 to 7-52
  - *default  2-12 to 2-13*
  - *overlay pages  7-49*
- map  2-13
  - *defined  D-5*
- model  7-24
- named  7-33 to 7-34
- pool, C language  7-10 , 7-83
- unconfigured  7-25
- widths, ordering memory words  11-13 to 11-14
- word ordering  11-13 to 11-14

MEMORY directive  7-24 to 7-28
- default model  7-24 , 7-51 to 7-52
- described  2-11
- syntax  7-24 to 7-28

memory map, overlaying pages  7-46

memory widths
- memory width (memwidth)  11-9 to 11-10
  - *exceptions  11-9*
- ROM width (romwidth)  11-10 to 11-12
- target width  11-9

−memwidth hex conversion utility option  11-4 , 11-9

memwidth ROMS specification  11-9 , 11-16

.mexit directive  5-3

−mf assembler option  3-6

−mg assembler option 3-6 , 3-39
.mlib directive 4-17 , 5-13
  reference 4-59
  use in macros 3-7
.mlist directive 4-15
  listing control 4-15 , 4-37
  reference 4-61
  use in macros 5-19
.mmregs directive, compatibility with C1x/C2x/C2xx/
  C5x 4-7
.mmsg directive 4-21 , 5-17
  listing control 4-15 , 4-37
  reference 4-38
mnemonic defined D-5
mnemonic field 3-12
.mnolist directive 4-15
  listing control 4-15 , 4-37
  reference 4-61
  use in macros 5-19
model statement 5-3
  defined D-5
Motorola-S object format 11-1 , 11-42

**N**

−n name utility option 10-16
name MEMORY specification 7-27
name utility
  invoking 10-16
  options 10-16
named memory 7-33 to 7-34
named sections 2-6 to 2-7 , A-3
  defined D-6
  .sect directive 2-7 , 4-69
  .usect directive 2-7 , 4-80
nested macros 5-21 to 5-22
.newblock directive 4-21 , 4-64
nm55 command 10-16
nm55 utility, invoking 10-16
.nolist directive 4-15 , 4-54
NOLOAD section 7-50

**O**

−o
  name utility option 10-16
  object file display option 10-2
−o option
  hex conversion utility 11-4
  linker 7-14
object code (source listing) 3-32
object file
  defined D-6
  library 7-22 to 7-23
object file display utility
  invoking 10-2
  options
    *−g 10-2*
    *−o 10-2*
    *−x 10-2*
object formats
  address bits 11-39
  ASCII-Hex 11-1 , 11-40
    *selecting 11-5*
  Intel 11-1 , 11-41
    *selecting 11-5*
  Motorola-S 11-1 , 11-42
    *selecting 11-5*
  output width 11-39
  Tektronix 11-1 , 11-44
    *selecting 11-5*
  TI-Tagged 11-1 , 11-43
    *selecting 11-5*
object libraries
  altering linker search algorithm 7-10 to 7-12
  defined D-6
  described 7-22 to 7-23
  run-time support 7-82
  using the archiver to build 6-2
octal integer constants 3-14
ofd6x command 10-2
on-chip boot loader
  boot table 11-28 to 11-37
  booting from device peripheral 11-31
  description 11-28 , 11-32 to 11-34
  modes 11-32
  options, summary 11-29
  setting the entry point 11-31
  using the boot loader 11-32 to 11-34
operands
  defined D-6
  field 3-12
  label 3-18
  local label 3-18 to 3-20
  source statement format 3-12

operator precedence order 3-26

.option directive 4-15 to 4-16 , 4-65

optional file header A-5
- defined D-6

options
- absolute lister 8-3
- archiver 6-4
- assembler 3-4 , 10-2
- cross-reference lister 9-3
- defined D-6
- hex conversion utility 11-3 to 11-4
- linker 7-5 to 7-18
- name utility 10-16
- strip utility 10-17

−order hex conversion utility option 11-13

ordering memory words 11-13 to 11-14

origin MEMORY specification 7-27

origin ROMS specification 11-15

output
- assembler 3-1
- executable 7-6
  - *relocatable 7-7*
- linker 7-3 , 7-14 , 7-86
- listing
  - *directive listing 4-37*
  - *enable 4-54*
  - *false conditional block listing 4-41*
  - *list options 4-65*
  - *macro listing 4-59 , 4-61*
  - *page eject 4-67*
  - *page length 4-53*
  - *page width 4-53*
  - *substitution symbol listing 4-72*
  - *suppress 4-54*
  - *tab size 4-77*
  - *title 4-79*
- module, defined D-6
- module name (linker) 7-14
- sections
  - *allocation 7-32 to 7-38*
  - *defined D-6*
  - *displaying a message 7-16*
  - *methods 7-51 to 7-52*

output listing 4-15 to 4-16

overflow (in expression) 3-26

overlay pages
- described 7-46
- example 7-47
- memory allocation 7-49
- using with sections 7-48

overlaying sections 7-43 to 7-44
- managing linker-generated copy tables 7-77 to 7-79

overriding assembler mode with directives 4-31

## P

−p, name utility option 10-16 , 10-17

page
- address specification 7-35
- eject 4-67
- length 4-53
- title 4-79
- width 4-53

.page directive 4-16 , 4-67

page MEMORY specification 7-27

parentheses in expressions 3-25

partial linking 7-80 to 7-81
- defined D-6

.port directive, compatibility with C1x/C2x/C2xx/C5x 4-7

precedence groups 3-25

predefined names
- −ad assembler option 3-4
- undefining with −au assembler option 3-5

−priority linker option 7-17

processor symbols 3-23

.pstring directive
- compatibility with C1x/C2x/C2xx/C5x 4-7
- reference 4-73

## Q

−q, name utility option 10-16

−q option
- absolute lister 8-3
- archiver 6-5
- assembler 3-6
- cross-reference lister 9-3
- hex conversion utility 11-4 , 11-6
- linker 7-15

quiet run
- assembler 3-6
- defined D-6
- linker 7-15

R

−r, name utility option 10-16
r archiver command 6-4
−r linker option 7-7 , 7-80 to 7-81
R MEMORY attribute 7-27
recursive macros 5-21 to 5-22
.ref directive 4-17
  identifying external symbols 2-18
  reference 4-46
relational operators, in conditional expressions 3-27
relocatable output module 7-7
  executable 7-7
relocation 2-14 to 2-16 , 7-6 to 7-7
  at run time 2-16
  capabilities 7-6 to 7-7
  defined D-6
  information A-9 to A-10
reserved words, linker 7-21
resetting local labels 4-64
ROM width (romwidth) 11-10 to 11-12
romname ROMS specification 11-15
ROMS directive 11-15 to 11-20
  creating map file of 11-19 to 11-20
  example 11-18 to 11-20
  parameters 11-15 to 11-16
−romwidth hex conversion utility option 11-4 , 11-11
romwidth ROMS specification 11-11 , 11-16
rts2800.lib 7-82 , 7-85
run address of a section 7-39
run linker keyword 2-16 , 7-39
RUN_START() linker operator 7-66
run-time
  initialization 7-82
  support 7-82

S

−s option
  archiver 6-5
  linker 7-15 , 7-80
.sblock directive 4-68
−sci8 hex conversion utility option 11-5 , 11-29 , 11-31 , 11-32

search libraries
  using −priority linker option 7-17
  using alternate mechanism 7-17
.sect directive 2-4
  described 4-8
  reference 4-69
.sect section 4-8
section
  allocation into memory 7-51 to 7-52
  COFF 2-1 to 2-20
  creating your own 2-6 to 2-7
  default allocation 7-51 to 7-52
  defined D-7
  description 2-2 to 2-3
  directives, default 2-4
  header A-6 to A-8
    *defined D-7*
  initialized 2-5 to 2-6
  input sections 7-30
  named 2-2 , 2-6 to 2-7
  number A-15
  overlaying with UNION statement 7-43 to 7-44
  relocation 2-14 to 2-16
    *at run time 2-16*
  special types 7-50
  specification 7-29
  specifying a run-time address 7-39
  specifying linker input sections 7-36 to 7-38
  uninitialized 2-4 to 2-5
    *initializing 7-64*
    *specifying a run address 7-40*
SECTIONS directive 7-29 to 7-38
  alignment 7-34
  allocation 7-32 to 7-38
  binding 7-33
  blocking 7-34
  default, allocation 7-51 to 7-52
  fill value 7-30
  GROUP 7-45
  input sections 7-30 , 7-36 to 7-38
  .label directive 7-40 to 7-42
  load allocation 7-29
  memory 7-33 to 7-34
  named memory 7-33 to 7-34
  reserved words 7-21
  run allocation 7-30
  section specification 7-29
  section type 7-30
  specifying
    *run-time address 2-16 , 7-39 to 7-42*

*two addresses 2-16 , 7-39*
syntax 7-29 to 7-30
uninitialized sections 7-40
UNION 7-43 to 7-45
use with MEMORY directive 7-24
SECTIONS hex conversion utility directive 11-21 to 11-22
parameters 11-21 to 11-22
SECTIONS linker directive, described 2-11
.set directive 4-19 , 4-70
.setsect assembler directive 8-8
.setsym assembler directive 8-8
.short directive 4-11
sign-extend, defined D-7
SIZE() linker operator 7-66
sname SECTIONS specification 11-21
source file
assembler 10-2
defined D-7
directory 3-7 to 3-9
source listings 3-31 to 3-33
source statement
field (source listing) 3-32
format 3-10
*comment field 3-12*
*label field 3-11*
*mnemonic field 3-12*
*operand field 3-12*
number (source listing) 3-31 to 3-33
.space directive 4-10 , 4-71
SPC (section program counter) 2-8
aligning
*by creating a hole 7-61*
*to word boundaries 4-13 to 4-14 , 4-22*
assembler’s effect on 2-8 to 2-10
assigning label 3-11
defined D-7
linker symbol 7-53 to 7-54 , 7-61
predefined symbol for 3-22
value
*associated with labels 3-11*
*shown in source listings 3-31*
special section types 7-50
special symbols in the symbol table A-12
−spi8 hex conversion utility option 11-5 , 11-29 , 11-31 , 11-32
−spibrr hex conversion utility option 11-5 , 11-29
.sslist directive 4-16
listing control 4-15 , 4-37
reference 4-72
use in macros 5-20
.ssnolist directive 4-16
listing control 4-15 , 4-37
reference 4-72
use in macros 5-20
__SSTACK_SIZE 7-56
−stack linker option 7-15
.stack section 7-15 , 7-83
__STACK_SIZE 7-15
.stag directive 4-74
stag structure tag 4-19
static symbols, creating with −h option 7-10
static variables A-11
defined D-7
status registers 3-23
storage classes A-14
defined D-7
.string directive 4-11
compatibility with C1x/C2x/C2xx/C5x 4-7
limiting listing with the .option directive 4-15 , 4-65
reference 4-73
string functions (substitution symbols)
$firstch 5-8
$iscons 5-8
$isdefed 5-8
$ismember 5-8
$isname 5-8
$isreg 5-8
$lastch 5-8
$symcmp 5-8
$symlen 5-8
string table A-13
defined D-7
strip utility
invoking 10-17
option 10-17
strip6x utility, invoking 10-17
stripping
line number entries 7-15
symbolic information 7-15
.struct directive 4-19 , 4-74
structure
.tag 4-74
defined D-7

stag   4-19

subsection
- defined   D-7
- initialized   2-7
- overview   2-7
- uninitialized   2-8

substitution symbols   3-24
- arithmetic operations on   4-19 , 5-7
- as local variables in macros   5-12
- assigning character strings to   3-24 , 4-19
- built-in functions   5-8 to 5-9
- directives that define   5-6
- expansion listing   4-16 , 4-72
- forced substitution   5-10
- in macros   5-5 to 5-12
- maximum number per macro   5-5
- passing commas and semicolons   5-5
- recursive substitution   5-9
- subscripted substitution   5-11
- .var directive   5-12

symbol
- assembler-defined   2-18 to 2-20 , 3-4
- assembler usage   3-18 to 3-24
- assigning values to
  - *at link time   7-53 to 7-60*
  - *reference   4-70*
- attributes   3-36
- case sensitivity   3-4
- character strings   3-17
- cross-reference lister   9-5
- defined   D-7
- defined only for C support   7-56
- definitions (cross-reference list)   3-36
- external   2-18 , 4-46
- global   7-10
- in a COFF file   2-18 to 2-20
- linker-defined   7-56
- names   A-13
- number of statements that reference   3-36
- predefined   3-22
- reserved words   7-21
- retaining duplicate information   7-8
- setting  to a constant value   3-21
- statement number that defines   3-36
- substitution   3-24
- table   2-19
  - *creating entries   2-19*
  - *defined   D-8*
  - *index   A-9*
  - *placing unresolved symbols in   7-16*
  - *special symbols used in   A-12*
  - *stripping entries   7-15*
  - *structure and content   A-11 to A-16*
  - *symbol values   A-14*
- undefining assembler-defined symbols   3-5
- unresolved   7-16
- used as labels   3-18
- value assigned   3-36

symbolic constants   3-22
- $   3-22
- defining   3-21
- processor symbols   3-23
- status registers   3-23

symbolic debugging   B-1 to B-14
- −as assembler option   3-5
- C-type for assembly variables   3-6
- defined   D-8
- directives   B-1 to B-14
- producing error messages in macros   5-17
- stripping symbolic information   7-15

syntax of assignment statements   7-53

__SYSMEM_SIZE   7-10

system stack, C language   7-15 , 7-83

## T

−t, name utility option   10-16

t archiver command   6-4

−t option, hex conversion utility   11-5 , 11-43

.tab directive   4-16 , 4-77

table() linker operator   7-69
- used to manage object components   7-70

.tag directive   4-19 , 4-74

target memory
- configuration   7-19
- defined   D-8
- loading a program into   7-9
- model   7-24

target width   11-9

Tektronix object format   11-1 , 11-44

.text directive   2-4
- described   4-8
- reference   4-78

.text linker symbol   7-56

.text section   4-8 , 4-78 , A-3
- defined   D-8

TI-Tagged object format   11-1 , 11-43

.title directive   4-16 , 4-79

TMS320C27x object mode 3-41
TMS320C28x assembler modes 3-41 to 3-46
TMS320C28x object −− accept C27x syntax mode 3-43
TMS320C28x object −− accept C2xLP syntax mode 3-43
TMS320C28x object mode 3-42

U

−u, name utility option 10-16
u archiver command 6-5
−u option, linker 7-16
unconfigured memory 7-25
  defined D-8
  overlaying 7-50
underflow (in expression) 3-26
uninitialized sections 2-4 to 2-5 , 7-61
  .bss section 2-4 , 4-27
  defined D-8
  initialization of 7-64
  specifying a run address 7-40
  subsections 2-5 , 2-8
  .usect section 2-4 , 4-80
UNION statement 7-43 to 7-45
  defined D-8
  memory overlay example 7-67
.usect directive 2-4 , 4-8
  compatibility with C1x/C2x/C2xx/C5x 4-7
  reference 4-80

V

−v archiver option 6-5
−v28 assembler options 3-4 , 3-41
.var directive 5-12
  listing control 4-15 , 4-37
variables, local, substitution symbols used as 5-12

W

−w linker option 7-16
W MEMORY attribute 7-27
well-defined expressions 3-27
  defined D-8
.width directive 4-16
  listing control 4-15
  reference 4-53
.wmsg directive 4-21 , 5-17
  listing control 4-15 , 4-37
word
  alignment 4-22
  defined D-8
.word directive 4-11
  limiting listing with the .option directive 4-15 to 4-16 , 4-65
  reference 4-51

X

x archiver command 6-4
X MEMORY attribute 7-27
−x object file display option 10-2
−x option
  assembler, cross-reference listing 3-36
  hex conversion utility 11-5 , 11-44
  linker 7-17
.xfloat directive 4-11
  compatibility with C1x/C2x/C2xx/C5x 4-7
  reference 4-45
.xlong directive
  compatibility with C1x/C2x/C2xx/C5x 4-7
  reference 4-56
xref2000 command 9-3
.xstring directive 4-11

Z

−zero hex conversion utility option 11-4
  defined D-1
  reference 4-22