Texas Instruments

TMS320C30 Assembly Language Tools

User's Guide

1988

Digital Signal Processing Products

User's Guide

TMS320C30 Assembly Language Tools

1988

TMS320C30 Assembly Language Tools User's Guide

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

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

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Copyright © 1988, Texas Instruments Incorporated

Contents

Section		Page
1	**Introduction**	**1-1**
1.1	Software Development Tools Overview	1-2
1.2	Getting Started	1-4
1.3	Manual Organization	1-5
1.4	Related Documentation	1-6
1.5	Style and Symbol Conventions	1-7
2	**Software Installation**	**2-1**
2.1	Installation for PCs	2-2
2.2	Installation for VAX/VMS	2-3
2.3	Installation for UNIX Systems	2-4
3	**Introduction to Common Object File Format**	**3-1**
3.1	Sections	3-2
3.2	How the Assembler Handles Sections	3-3
3.2.1	Uninitialized Sections	3-3
3.2.2	Initialized Sections	3-4
3.2.3	Named Sections	3-5
3.2.4	Section Program Counters	3-6
3.2.5	Absolute Sections	3-6
3.2.6	An Example That Uses Sections Directives	3-6
3.3	How the Linker Handles Sections	3-10
3.3.1	Default Allocation	3-10
3.3.2	Placing Sections in the Memory Map	3-13
3.4	Relocation	3-15
3.5	Loading a Program	3-16
3.6	Symbols in a COFF File	3-17
3.6.1	External Symbols	3-17
3.6.2	The Symbol Table	3-17
4	**Assembler Description**	**4-1**
4.1	Assembler Development Flow	4-2
4.2	Invoking the Assembler	4-3
4.3	Specifying Alternate Directories for Assembler Input	4-4
4.3.1	-i Assembler Option	4-4
4.3.2	Environment Variable (A_DIR)	4-5
4.4	Source Statement Format	4-6
4.4.1	Label Field	4-6
4.4.2	Mnemonic Field	4-7
4.4.3	Operand Field	4-7
4.4.4	Comment Field	4-7
4.5	Constants	4-8
4.5.1	Binary Integers	4-8
4.5.2	Octal Integers	4-8
4.5.3	Decimal Integers	4-9
4.5.4	Hexadecimal Integers	4-9
4.5.5	Character Constants	4-9
4.5.6	Floating-Point Constants	4-10

4.5.7 Assembly-Time Constants . . . 4-10
4.6 Character Strings . . . 4-11
4.7 Symbols . . . 4-11
4.8 Expressions . . . 4-12
4.8.1 Operators . . . 4-12
4.8.2 Expression Overflow or Underflow . . . 4-13
4.8.3 Well-Defined Expressions . . . 4-13
4.8.4 Conditional Expressions . . . 4-13
4.8.5 Relocatable Symbols and Legal Expressions . . . 4-13
4.9 Source Listings . . . 4-15
4.10 Cross-Reference Listings . . . 4-17

5 Assembler Directives **5-1**
5.1 Directives Summary . . . 5-2
5.2 Sections Directives . . . 5-4
5.3 Directives that Initialize Memory . . . 5-6
5.4 Directives that Align the Section Program Counter . . . 5-9
5.5 Directives that Format the Output Listing . . . 5-10
5.6 Conditional Assembly Directives . . . 5-11
5.7 Directives that Reference Other Files . . . 5-12
5.8 Directives Reference . . . 5-13

6 Instruction Set **6-1**
6.1 Summary . . . 6-2
6.1.1 Addressing Modes . . . 6-2
6.1.2 Optional Syntaxes . . . 6-3
6.1.3 Condition Codes . . . 6-4
6.1.4 Instruction Set Summary Table . . . 6-5
6.2 Three-Operand Instructions . . . 6-17
6.3 Parallel Instructions . . . 6-18
6.4 Load and Store Instructions . . . 6-21
6.5 Arithmetic Instructions . . . 6-22
6.6 Logical Instructions . . . 6-22
6.7 Program-Control Instructions . . . 6-23
6.8 Interlocked-Operation Instructions . . . 6-23
6.9 The LDP Instruction . . . 6-24

7 Macro Language **7-1**
7.1 Macro Directives Summary . . . 7-2
7.2 Macro Libraries . . . 7-3
7.3 Defining Macros . . . 7-4
7.4 Macro Parameters . . . 7-6
7.5 Conditional Blocks . . . 7-7
7.6 Repeatable Blocks . . . 7-8
7.7 Unique Labels . . . 7-9

8 Archiver Description 8-1
8.1 Archiver Development Flow 8-2
8.2 Invoking the Archiver 8-3
8.3 Archiver Examples 8-4

9 Linker Description 9-1
9.1 Linker Development Flow 9-2
9.2 Invoking the Linker 9-3
9.3 Linker Options 9-4
9.3.1 Relocation Capability (-a and -r Options) 9-4
9.3.2 C Language Options (-c and -cr Options) 9-6
9.3.3 Define an Entry Point (-e symbol Option) 9-6
9.3.4 Set Default Fill Value (-f cc Option) 9-6
9.3.5 Make All Global Symbols Static (-h Option) 9-7
9.3.6 Alter the Library Search Algorithm (-i dir and -l filename Options/C—DIR 9-7
9.3.7 Create a Map File (-m filename Option) 9-9
9.3.8 Name an Output Module (-o filename Option) 9-10
9.3.9 Specify a Quiet Run (-q Option) 9-10
9.3.10 Strip Symbolic Information (-s Option) 9-10
9.3.11 Introduce an Unresolved Symbol (-u symbol Option) 9-10
9.4 Linker Command Files 9-11
9.5 Object Libraries 9-13
9.6 The MEMORY Directive 9-14
9.6.1 Default Memory Model 9-14
9.6.2 MEMORY Directive Syntax 9-14
9.7 The SECTIONS Directive 9-16
9.7.1 Default Sections Configuration 9-16
9.7.2 SECTIONS Directive Syntax 9-16
9.7.3 Specifying Input Sections 9-18
9.7.4 Specifying the Address of an Output Section (Allocation) 9-20
9.7.5 Grouping Output Sections Together 9-22
9.8 Overlay Pages 9-23
9.8.1 Using the MEMORY Directive to Define Overlay Pages 9-23
9.8.2 Using Overlay Pages with the SECTIONS Directive 9-24
9.8.3 Page Definition Syntax 9-25
9.9 Default Allocation 9-27
9.9.1 Allocation Algorithm 9-27
9.9.2 General Rules for Output Sections 9-27
9.10 Special Section Types (DSECT, COPY, and NOLOAD) 9-29
9.11 Assigning Symbols at Link Time 9-30
9.11.1 Syntax of Assignment Statements 9-30
9.11.2 Assigning the SPC to a Symbol 9-30
9.11.3 Assignment Expressions 9-31
9.11.4 Symbols Defined by the Linker 9-32
9.12 Creating and Filling Holes 9-33
9.12.1 Initialized and Uninitialized Sections 9-33
9.12.2 Creating Holes 9-33
9.12.3 Filling Holes 9-35
9.12.4 Explicit Initialization of Uninitialized Sections 9-36
9.13 Partial (Incremental) Linking 9-37
9.14 Linking C Code 9-38
9.14.1 Runtime Initialization 9-38
9.14.2 Object Libraries and Runtime Support 9-38
9.14.3 Autoinitialization (ROM and RAM Models) 9-38

9.14.4 The -c and -cr Linker Options 9-40
9.15 Linker Example 9-41

10 Object Format Converter Description **10-1**
10.1 Object Format Converter Development Flow 10-2
10.2 Invoking the Object Format Converter 10-3
10.3 Examples 10-4
10.4 Halt Conditions 10-4

A Common Object File Format **A-1**
B Symbolic Debugging Directives **B-1**
C Assembler Error Messages **C-1**
D Linker Error Messages **D-1**
E ASCII Character Set **E-1**
F Glossary **F-1**

Illustrations

Figure		*Page*
1-1	TMS320C30 Assembly Language Development Flow	1-2
3-1	Partitioning Memory into Logical Blocks	3-2
3-2	Using Sections Directives	3-8
3-3	Object Code Generated by Figure 3-2	3-9
3-4	Placing the Object Code from Figure 3-2 into Memory (Default Allocation)	3-11
3-5	Combining Input Sections from Two Files (Default Allocation)	3-12
3-6	MEMORY and SECTIONS Directives for Figure 3-7	3-13
3-7	Rearranging the Memory Map from Figure 3-4	3-14
3-8	An Example of Code that Generates Relocation Entries	3-15
4-1	Assembler Development Flow	4-2
4-2	Sample Assembler Listing	4-16
4-3	Cross-Reference Listing Format	4-17
5-1	Examples of Sections Directives	5-5
5-2	Examples of Initialization Directives	5-7
5-3	An Example of the .field Directive	5-7
5-4	An Example of the .space Directive	5-8
5-5	An Example of the .align Directive	5-9
5-6	An Example of the .even Directive	5-9
5-7	An Example of Conditional Assembly Directives	5-11
5-8	An Example of the .align Directive	5-14
5-9	An Example of the .asect Directive	5-16
5-10	An Example of the .even Directive	5-24
5-11	An Example of the .field Directive	5-27
5-12	An Example of the .space Directive	5-43
5-13	An Example of the .usect Directive	5-48
7-1	An Example of a Conditional Block	7-7
8-1	Archiver Development Flow	8-2
9-1	Linker Development Flow	9-2
9-2	An example of a Linker Command File	9-11
9-3	An Example of a Command File with Linker Directives	9-12
9-4	An Example of the MEMORY Directive	9-14
9-5	Memory Map Defined in Figure 9-4	9-15
9-6	An Example of the SECTIONS Directive	9-16
9-7	Section Allocation Defined by Figure 9-6	9-18
9-8	The Most Common Method of Specifying Section Contents	9-18
9-9	Overlay Page Example	9-24
9-10	ROM Model of Autoinitialization	9-39
9-11	RAM Model of Autoinitialization	9-40
9-12	Linker Command File, demo.cmd	9-42
9-13	Output Map File, demo.map	9-43
10-1	Object Format Converter Development Flow	10-2
A-1	COFF File Structure	A-2
A-2	Sample COFF Object File	A-3
A-3	An Example of Section Header Pointers for the .text Section	A-7
A-4	Line Number Blocks	A-9
A-5	Line Number Entries Example	A-10
A-6	Symbol Table Contents	A-11
A-7	Symbols for Blocks	A-13

A-8 Symbols for Functions A-13
A-9 Sample String Table A-14

Tables

Table *Page*

4-1 Operators 4-12
4-2 Expressions with Absolute and Relocatable Symbols 4-13
4-3 Symbol Attributes for Cross-Reference Listings 4-17
5-1 Directives Summary 5-2
6-1 Indirect Addressing Mode 6-3
6-2 Condition Codes 6-4
6-3 Summary Three-Operand Instructions 6-17
6-4 Summary of Parallel Instructions 6-19
6-5 Summary of Load and Store Instructions 6-21
6-6 Summary of Arithmetic Instructions 6-22
6-7 Summary of Logical Instructions 6-22
6-8 Summary of Program-Control Instructions 6-23
6-9 Summary of Interlocked-Operation Instructions 6-23
9-1 Linker Options Summary 9-4
9-2 Operators in Assignment Expressions 9-32
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 36 and 37) A-6
A-6 Relocation Entry Contents A-8
A-7 Relocation Types (Bytes 8 and 9) A-8
A-8 Line Number Entry Format A-9
A-9 Symbol Table Entry Contents A-12
A-10 Special Symbols in the Symbol Table A-12
A-11 Symbol Storage Classes A-15
A-12 Special Symbols and Their Storage Classes A-15
A-13 Symbol Values and Storage Classes A-16
A-14 Section Numbers A-17
A-15 Basic Types A-18
A-16 Derived Types A-18
A-17 Auxiliary Symbol Table Entries Format A-19
A-18 Section Format for Auxiliary Table Entries A-19
A-19 Section Format for Auxiliary Table Entries A-20
A-20 Tag Name Format for Auxiliary Table Entries A-20
A-21 End of Structure Format for Auxiliary Table Entries A-20
A-22 Function Format for Auxiliary Table Entries A-21
A-23 Array Format for Auxiliary Table Entries A-21
A-24 End of Blocks and Functions Format for Auxiliary Table Entries A-21
A-25 Beginning of Blocks and Functions Format for Auxiliary Table Entries A-22
A-26 Structure, Union, and Enumeration Names Format for Auxiliary Table Entries A-22

Section 1

Introduction

The TMS320C30 Digital Signal Processor is an advanced CMOS 32-bit microprocessor that is optimized for signal processing applications. The TMS320C30 is the third generation in the Texas Instruments family of digital signal processors.

The TMS320C30 is well supported by a full set of hardware and software development tools, including a C compiler, a full-speed in-circuit emulator, and a software simulator. This document discusses the software development tools that are included with the TMS320C30 assembly language package:

- Assembler
- Archiver
- Linker
- Object format converter

These tools can be installed on the following systems:

- IBM-PC/PC-DOS and compatibles
- VAX/VMS (revisions 3.7 and up)
- VAX/Ultrix
- Sun-3 Workstations with UNIX

The TMS320C30 assembly language tools create and use object files that are in common object file format, or **COFF**. COFF object files contain separate blocks (called *sections*) of code and data that you can load into different TMS320C30 memory spaces. You will be able to program the TMS320C30 more efficiently if you have a basic understanding of COFF; Section 3, Introduction to Common Object File Format, discusses this object format in detail.

Topics covered in this introductory section include:

Section		Page
1.1	Software Development Tools Overview	1-2
1.2	Getting Started	1-4
1.3	Manual Organization	1-5
1.4	Related Documentation	1-6
1.5	Style and Symbol Conventions	1-7

1.1 Software Development Tools Overview

Figure 1-1 shows the TMS320C30 assembly language development flow. The center section of the illustration highlights the most common path; the other portions are optional.

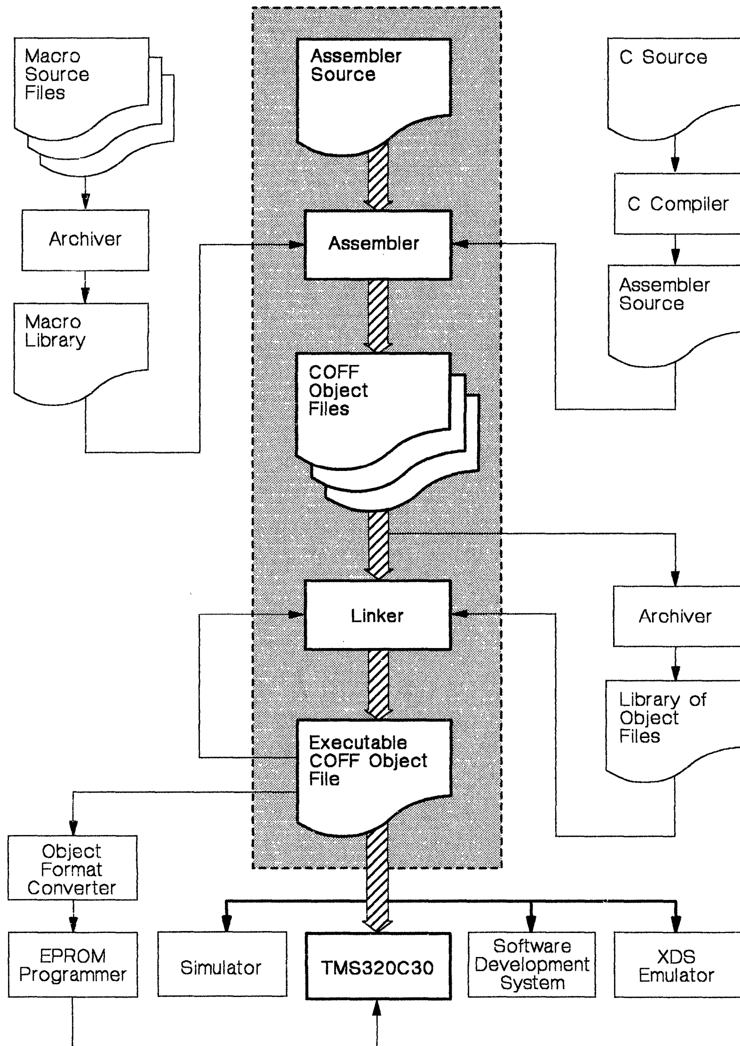


Figure 1-1. TMS320C30 Assembly Language Development Flow

- The **C compiler** translates C source code into TMS320C30 assembly language source code. The C compiler is not included as part of the assembly language tools package.

- The **assembler** translates assembly language source files into machine language 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.

- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files (created by the assembler) as input. It can also accept archive 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 specific addresses or within memory ranges, and define or redefine global symbols.

- The **archiver** allows you to collect a group of files into a single archive library. Both the assembler and linker can use archive libraries as input. For example, you could collect several macros together into a macro library; the assembler can search through a library and use the members that the source file calls as macros. You could use the archiver to collect a group of object files into an object library; the linker can link in the library members that resolve external references.

- The main purpose of this development process is to produce a module that can be executed in a system that contains a **TMS320C30**. You can use one of several debugging tools to refine and correct your code before downloading it to a TMS320C30 system. These debugging tools share a common screen-oriented interface that displays and maintains machine status information and controls execution of the system that is being developed. Note that only *linked* object files can be executed.

  - The **simulator** is a software program that simulates TMS320C30 functions. The simulator can execute linked COFF object modules. The simulator is not included with the TMS320C30 assembly language package.

  - The **XDS** (extended development support) **emulator** is a realtime, in-circuit emulator with the same screen-oriented interface as the software simulator. The emulator is not included with the TMS320C30 assembly language package.

  - The **software development system** (SWDS) is a PC-resident tool that executes code on a TMS320C30. The SWDS is not included with the TMS320C30 assembly language package.

- Most EPROM programmers do not accept COFF object files as input. The **object format converter** converts a COFF object file into TI-tagged, Intel, or Tektronix object format. The converted file can be downloaded to an EPROM programmer.

1.2 Getting Started

The tools you will probably use most often are the assembler and the linker. This section provides a quick walkthrough so that you can get started without reading the whole user's guide. These examples show the most common methods for invoking the assembler and linker.

1) Create two short source files to use for the walkthrough; call them `filea.asm` and `fileb.asm`.

filea.asm	fileb.asm

<pre>   .file   "filea"
        .global addvec
vector .word   10,20,30,40
        LDI   vector,AR0
        CALL   addvec</pre>
	<pre> .file "fileb"
	        .global addvec
	addvec LDI 0,R0
	        RPTS 3
	        ADDI *AR0++,R0
	        RETS</pre>

2) Assemble `filea.asm`; enter:

```
asm30 filea
```

The asm30 command invokes the assembler. `filea.asm` is the input source file. (If the input file extension is .asm, you don't have to specify the extension; the assembler uses .asm as the default.) This example creates an object file called `filea.obj`. The assembler always creates an object file. You can specify a name for the object file, but if you don't, the assembler will use the input filename with an extension of .obj.

Now assemble `fileb.asm`; enter:

```
asm30 fileb -l
```

This time, the assembler creates an object file called `fileb.obj`. The -l (lowercase "L") option tells the assembler to create a listing file; the listing file for this example is called `fileb.lst`.

3) Link `filea.obj` and `fileb.obj`; enter:

```
lnk30 filea fileb -o prog.out
```

The lnk30 command invokes the linker. `filea.obj` and `fileb.obj` are the input object files. (If the input file extension is .obj, you don't have to specify the extension; the linker uses .obj as the default.) The linker combines `filea.obj` and `fileb.obj` to create an executable object module called `prog.out` (the -o option supplies the name of the output module).

You can find more information about invoking the tools in the following sections:

Section		Page
4.2	Invoking the Assembler	4-3
8.2	Invoking the Archiver	8-3
9.2	Invoking the Linker	9-3
10.2	Invoking the Object Format Converter	10-3

1.3 Manual Organization

Section 1 **Introduction**
Provides an overview of the assembly language tools and the assembly language development process, gives quick examples for invoking the tools, lists related documentation, and explains the style and symbol conventions used throughout this document.

Section 2 **Software Installation**
Contains instructions for installing the assembly language tools on VAX/VMS, VAX/Ultrix, Sun-3/UNIX, and IBM-PC/PC-DOS systems.

Section 3 **Introduction to Common Object File Format**
Discusses the basic COFF concept of **sections** and how they can help you to use the assembler and linker more efficiently. *Read Section 3 before using the assembler and linker.*

Section 4 **Assembler Description**
Tells you how to invoke the assembler and discusses source statement format, valid constants and expressions, and assembler output.

Section 5 **Assembler Directives**
Divided into two parts; the first part describes the directives according to function, and the second part is an alphabetical reference.

Section 6 **Instruction Set Summary**
Summarizes the TMS320C30 instruction set alphabetically and by function; also summarizes addressing modes and optional syntax forms.

Section 7 **Macro Language**
Describes macro directives and macro creation.

Section 8 **Archiver Description**
Contains instructions for invoking the archiver, creating new archive libraries, and modifying existing libraries.

Section 9 **Linker Description**
Tells you how to invoke the linker, provides details of linker operation, discusses linker directives, and presents a detailed linking example.

Section 10 **Object Format Converter Description**
Tells you how to invoke the object format converter so that you can convert a COFF object file into an Intel, Tektronix, or TI-tagged object format.

Appendix A **Common Object File Format**
Contains specific information about the internal format of COFF object files.

Appendix B **Symbolic Debugging Directives**
Lists the symbolic debugging directives that theTMS320C30 C compiler uses.

Appendix C **Assembler Error Messages**
Appendix D **Linker Error Messages**
List the assembler and linker error messages.

Appendix E **ASCII Character Set**
Provides a table of the ASCII character set.

Appendix F **Glossary**
Defines a glossary of terms and acronyms used in this book.

1.4 Related Documentation

The following TMS320C30 documents are available from Texas Instruments:

- The ***TMS320 Family Development Support Reference Guide*** (literature number SPRU011) describes the wide range of TMS320 products that are available.

- ***Details on Signal Processing*** is a quarterly newsletter that provides information about new TMS320 family products, new documentation, development tool updates, and similar information. If you would like your name added to the newsletter mailing list, call the Texas Instruments Customer Response Center (1-800-232-3200).

- The ***Third-Generation TMS320 User's Guide*** (literature number SPRU031) discusses hardware and software aspects of the TMS320C30, such as pin functions, architecture, and interfaces, and contains the TMS320C30 instruction set.

- The ***TMS320C30 C Compiler Reference Guide*** (literature number SPRU034) tells you how to use the TMS320C30 C compiler. This C compiler accepts standard Kernighan and Ritchie C source code and produces TMS320C30 assembly language source code. We suggest that you use *The C Programming Language* (written by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall) as a companion to the *TMS320C30 C Compiler Reference Guide.*

1.5 Style and Symbol Conventions

- In this document program listings, program examples, screen displays, filenames, and symbol names are shown in a `special font`. Examples use a **`bold version of the special font`** for emphasis. Here is a sample program listing:

```
0011  0005  0001          .field     1, 2
0012  0005  0003          .field     3, 4
0013  0005  0006          .field     6, 3
0014  0006                .even
```

- In a syntax description, the instruction, command, or directive is in a **bold face font** and parameters are in *italics*. Portions of a syntax that are in **bold face** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  .asect *"section name", address*

  .asect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- Square brackets ([and]) indicate an optional parameter. For example, the **asm30** command has several optional parameters:

  asm30 *[input file [object file [listing file]]] [-options]*

  - The first parameter, *input file*, is optional.
  - The second parameter, *object file*, is optional; in addition, you can specify an *object file* only if you also specified an *input file*.
  - The third parameter, *listing file*, is optional; in addition, you can specify a *listing file* only if you also specified an *input file* and an *object file*.
  - The fourth parameter, *-options*, is optional; you can specify options even if you specified no other parameters.

  Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they aren't optional).

- Braces ({ and }) indicate a list. The | symbol (read as *or*) separates items within a list. Here's an example of a list:

  { * | *+ | *- }

  This list provides three choices: *, *+, or *-.

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

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

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Section 2

Software Installation

This section contains step-by-step instructions for installing the assembler, archiver, linker, and object format converter. This software can be installed on the following systems:

- **DEC VAX/VMS**[1]
- **IBM-PC with PC-DOS**[2] (versions 2.1 and up) and compatibles
- **UNIX**[3] **Systems**
  - VAX/Ultrix
  - SUN-3

You will find installation instructions for these systems in the following sections:

Section **Page**

2.1 Installation for PCs 2-2
2.2 Installation for VAX/VMS 2-3
2.3 Installation for UNIX Systems 2-4

Section 1.5 (page 1-7) lists style and symbol conventions that are used in this section.

1 VAX and VMS are trademarks of Digital Equipment Corporation.

2 PC-DOS is a trademark of International Business Machines.

3 UNIX is a registered trademark of AT&T.

2.1 Installation for PCs

The TMS320C30 software package is shipped on two double-sided, double-density diskettes. The disk labelled ASM/LINK/ARCH contains the assembler, linker, and archiver. The disk labelled ROM/DEMO contains the object format converter. The tools execute in batch mode. At least 512K bytes of memory space must be available in your system.

These instructions are for both hard disk systems and dual floppy drive systems. On a dual-drive system, the PC-DOS system diskette should be in drive B. The instructions use these symbols for drive names:

A: Floppy disk drive for hard disk systems *or* source drive for dual-drive systems.
B: Destination or system disk drive for dual-drive systems.
C: Winchester (hard disk) for hard disk systems.

1) Make backups of the product diskettes.

2) Create a directory to contain the TMS320C30 software.

   - On *hard disk* systems, enter: `MD C:\C30TOOLS`
   - On *dual-drive* systems, enter: `MD B:\C30TOOLS`

3) Copy the TMS320C30 tools onto the hard disk or the system disk.

   - On *hard disk* systems, enter: `COPY A:*.* C:\C30TOOLS*.*`
   - On *dual-drive* systems, enter: `COPY A:*.* B:\C30TOOLS*.*`

2.2 Installation for VAX/VMS

The TMS320C30 software tape was created with the VMS BACKUP utility at 1600 BPI. These tools were developed on version 4.5 of VMS. If you are using an earlier version of VMS, you may need to relink the object files; refer to the Release Notes for relinking instructions.

1) Mount the tape on your tape drive.

2) Execute the following VMS commands. Note that you must create a destination directory for the tools; in this example, `DEST:directory` represents that directory. Replace `TAPE` with the name of the tape drive you are using.

```
$ allocate              TAPE:
$ init/den=1600         TAPE:C30
$ mount/for/den=1600    TAPE:
$ backup                TAPE:ASM30.bck DEST[:directory]
$ dismount              TAPE:
$ dealloc               TAPE:
```

3) The product tape contains a file called `setup.com`. This file sets up VMS symbols that allow you to execute the tools in the same manner as other VMS commands. Enter the following command to execute the file:

```
$ @setup                DEST:directory
```

This sets up symbols that you can use to call the various tools. As the file is executed, it will display the defined symbols on the screen.

You may want to include the commands from `setup.com` in your `login.com` file. This automatically defines symbols for running the tools each time you log in.

2.3 Installation for UNIX Systems

The TMS320C30 product tape was made at 1600 BPI using *tar* utility. Follow these instructions to install the assembly language tools package:

1) Mount the tape on your tape drive.
2) Make sure that the directory that you'll store the tools in is the current directory.
3) Enter the *tar* command for your system; for example,

   ```
   tar x
   ```

   This copies the entire tape into the directory.

Section 3

Introduction to Common Object File Format

The assembler and linker create object files that can be executed by the TMS320C30. The format that these object files are in is called *common object file format*, or ***COFF***.

COFF object format 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 provides an overview of COFF sections and includes the following topics:

Section		Page
3.1	Sections	3-2
3.2	How the Assembler Handles Sections	3-3
3.3	How the Linker Handles Sections	3-9
3.4	Relocation	3-14
3.5	Loading a Program	3-15
3.6	Symbols in a COFF File	3-16

Appendix A details COFF object structure; for example, it describes the fields in a file header and the structure of a symbol table entry. Appendix A is mainly useful for those of you who are interested in the internal format of object files.

3.1 Sections

The smallest relocatable unit of an object file is called a **section**. A section is a relocatable block of code or data which will (ultimately) occupy contiguous space in TMS320C30 memory. Each section of an object file is separate and distinct from the other sections. COFF object files always contain three default sections:

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

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

It is important to note that 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 and .asect assembler directives 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.

The assembler provides several directives that 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 that is organized similarly to the object file shown in Figure 3-1.

One of the linker's functions is to relocate sections into the target memory map (this is called **allocation**). Since most systems contain several different types of memory, using sections can help you to use target memory more efficiently. All sections are independently relocatable; you can place different sections into various blocks of target memory. For example, you can define a section that contains an initialization routine, and then allocate the routine into the portion of the memory map that contains EPROM.

Figure 3-1 shows the relationship between sections in an object file and a hypothetical target memory.

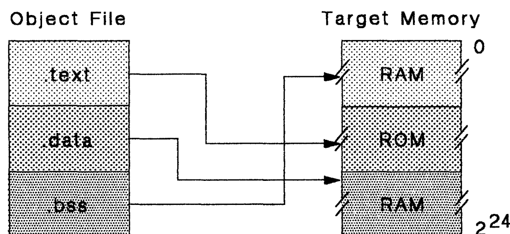


Figure 3-1. Partitioning Memory into Logical Blocks

3.2 How the Assembler Handles Sections

The assembler's main function in regard to sections is to identify the portions of an assembly language program that belong in a particular section. The assembler has six directives that support this function:

- The **.bss** and **.usect directives** reserve defined amounts of space in memory (usually RAM). This reserved space is used for storing variables.
- The **.text directive** identifies the source statements that follow it as executable code. The statements following a .text directive are assembled into the .text section.
- The **.data directive** identifies the source statements that follow it as initialized data. The statements following a .data directive are assembled into the .data section.
- The **.sect** and **.asect directives** define named sections that can be used like the .text and .data sections. The .sect directive creates a section with relocatable addresses; the .asect directive creates a section with absolute addresses. The statements following a .sect or .asect directive are assembled into the appropriate named section.

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

> **Note:**
>
> If you don't use any of the sections directives, the assembler assembles everything into the .text section.

3.2.1 Uninitialized Sections

Uninitialized sections reserve space in TMS320C30 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 directive, the assembler reserves more space in the .bss section. Each time you invoke the .usect directive, the assembler reserves more space in the specified named section.

You will usually allocate all variables into the .bss section. Occasionally, you may find it convenient to reserve additional space for variables and allocate this space separately from .bss; you can use .usect for this purpose.

The syntaxes for these directives are:

        .bss *symbol, size in words*
symbol **.usect** *"section name", size in words*

- The *symbol* points to the first word reserved by this invocation of the .bss or .usect directive. The *symbol* corresponds to the name of the variable that you're reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the .global assembler directive).
- The *size* is an absolute expression. The .bss directive reserves *size* words in the .bss section; the .usect directive reserves *size* words in section *name*.
- The *section name* parameter tells the assembler which named section to reserve space in. (For more information about named sections, see Section 3.2.3.)

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

3.2.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in TMS320C30 memory when the program is loaded. Each initialized section is separately relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section-relative references.

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

.text
.data
.sect *"section name"*
.asect *"section name", address*

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

Sections are built up 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, .sect, or .asect directive). If the assembler encounters subsequent .data directives, it *adds* the statements following these .data directives to the statements that are already in the .data section. This creates a single .data section that can be allocated contiguously into memory.

3.2.3 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 from the default sections.

For example, repeated use of the .text directive builds up a single .text section in the object file. When linked, this .text section is allocated into memory as a single unit. Suppose there is a portion of executable code (perhaps an initialization routine) that you don't want allocated with .text. If you assemble this segment of code into a named section, it is assembled separately from .text, and you will be able to allocate it into memory separately from .text. (Note that 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.)

Three directives let you create named sections:

- The **.usect** directive creates sections that are used like the .bss section. These sections reserve space in RAM for variables.
- The **.sect** and **.asect** directives create sections that can be used like the default .text and .data sections. The .sect directive creates named sections with *relocatable* addresses; the .asect directive creates named sections with *absolute* addresses.

The syntaxes for these directives are:

symbol **.usect** *"section name", size in words*
.sect *"section name"*
.asect *"section name", address*

The *section name* parameter is the name of the section. Section names are significant to 8 characters. You can create up to 32,767 separate named sections.

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

3.2.4 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 and 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 at that point.

The assembler treats each section as if it begins at address 0; the linker relocates each section according to its final location in the memory map.

3.2.5 Absolute Sections

The .asect directive defines a named section whose addresses are absolute with respect to a specified address. Absolute sections are useful for loading code from off-chip memory into faster on-chip memory.

The syntax for this directive is:

.asect *"section name", address*

The *section name* parameter identifies the name of the absolute section (Section 3.2.3 describes named sections). The *address* parameter identifies the section's absolute starting address in target memory. In order to use an absolute section, you must know which location you want the section to execute from, and specify it as the *address* parameter.

Most sections directives create sections with relocatable addresses. These sections always have an initial SPC value of 0; the linker relocates these sections appropriately. The initial SPC value for an absolute section, however, is the specified *address*. The addresses of all code assembled into an absolute section are offsets from this address. The linker **does** relocate sections defined with .asect; however, any labels defined within an absolute section retain their absolute (*runtime*) addresses. Thus, references to these labels refer to their *runtime* addresses, even though the section is not initially loaded at this address.

3.2.6 An Example That Uses Sections Directives

Figure 3-2 shows how you can build COFF sections incrementally, using the sections directives to swap back and forth between the different sections. You can use sections directives:

- To begin assembling code or data into a section for the first time, **or**
- To continue assembling into a section that already contains code. In this case, the assembler simply appends the new code to the code that is already assembled into the section.

The format of this example is a listing file. By using a listing file, this example shows how the SPCs are modified during assembly. A line in a listing file has four fields:

```
0001                          ***********************************************
0002                          **  Assemble an initialized table into .data **
0003                          ***********************************************
0004 000000                            .data
0005 000000 00000011          coeff    .word      011h, 022h, 033h
     000001 00000022
     000002 00000033
0006                          ***********************************************
0007                          **  Reserve space in .bss for two variables  **
0008                          ***********************************************
0009 000000                            .bss       var1,1
0010 000001                            .bss       buffer, 10
0011
0012                          ***********************************************
0013                          **  Still in .data                           **
0014                          ***********************************************
0015 000003 00000123          ptr      .word      0123h
0016
0017                          ***********************************************
0018                          **  Assemble code into the .text section     **
0019                          ***********************************************
0020 000000                            .text
0021 000000 0869000A          add:     LDI        10,AR1
0022 000001 08610000                   LDI        0,R1
0023 000002                   aloop:
0024 000002 02412001                   ADDI       *AR0++,R1
0025 000003 6E46FFFE                   DBNZ       AR1,aloop
0026 000004 15210000+                  STI        R1,@var1
0027
0028                          ***********************************************
0029                          ** Assemble another initialized table into   **
0030                          ** the .data section                         **
0031                          ***********************************************
0032 000004                            .data
0033 000004 000000AA          ivals    .word      0AAh, 0BBh, 0CCh
     000005 000000BB
     000006 000000CC
0034                          ***********************************************
0035                          ** Define another section for more variables **
0036                          ***********************************************
0037 000000                   var2     .usect     "newvars",1
0038 000001                   inbuf    .usect     "newvars",7
0039
0040                          ***********************************************
0041                          ** Assemble more code into .text             **
0042                          ***********************************************
0043 000005                            .text
0044 000005 0869000A          mpy:     LDI        10,AR1
0045 000006 08610000                   LDI        0,R1
0046 000007                   mloop:
0047 000007 0AC12001                   MPYI       *AR0++,R1
0048 000008 6E46FFFE                   DBNZ       AR1,mloop
0049 000009 15210007+                  STI        R1,@var2
0050
0051                          ***********************************************
0052                          ** Define a named section for int. vectors   **
0053                          ***********************************************
0054 000000                            .sect      "vectors"
0055 000000 00000000'                  .word      add, mpy
     000001 00000005'
```

Figure 3-2. Using Sections Directives

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

.text contains 10 words of object code.

.data contains 7 words of object code.

vectors is a named section created with the .sect directive; it contains 2 words of initialized data.

.bss reserves 11 words in memory.

newvars is a named section created with the .usect directive; it reserves 8 words in memory.

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

Line Numbers	Object Code
	.text section
25	0869000A
26	08610000
28	02412001
29	6E46FFFE
30	15210000
50	0869000A
51	0861000A
53	0AC12001
54	6E46FFFE
55	15210000
	.data section
6	00000011
6	00000022
6	00000033
18	00000123
37	000000AA
37	000000BB
37	000000CC
	vectors
62	00000000
62	00000005
	.bss
12, 13	No data– 11 words reserved
	newvars
43, 44	No data– 8 words reserved

Figure 3-3. Object Code Generated by Figure 3-2

3.3 How the Linker Handles Sections

The linker has two main functions in regard 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.

The linker provides two directives that 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 and where to place the output sections in memory.

It is not always necessary to use linker directives. If you don't use them, the linker uses the default allocation algorithm described in Section 3.3.1. When you *do* use linker directives, you must specify them in a linker command file.

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

Section		Page
9.4	Linker Command Files	9-11
9.6	The MEMORY Directive	9-14
9.7	The SECTIONS Directive	9-16

3.3.1 Default Allocation

You can link files without specifying a MEMORY or SECTIONS directive. The linker uses a default model to combine sections (if necessary) and allocate them into memory. When using the default model, the linker:

1) Assumes that memory begins at address 0h.
2) Assumes that 2^{24} words are available to allocate object code into.
3) Allocates the .text section into memory, beginning at address 0.
4) Allocates the .data section into memory, immediately following .text.
5) Allocates the .bss section into memory, immediately following .data.
6) Allocates *all* named sections into memory, immediately following .bss. Named sections are allocated in the order that they're encountered in the input files.

Note that the linker does not actually place an object code into memory; it assigns addresses to sections so that a *loader* can place the code in memory.

Figure 3-4 shows how a *single* file would be allocated using default allocation.

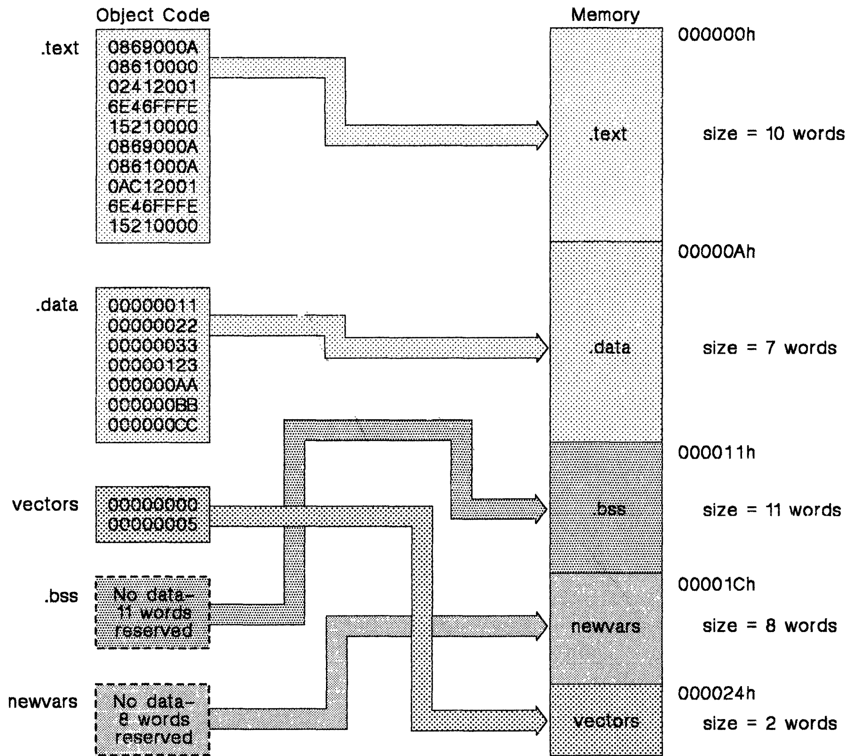


Figure 3-4. Placing the Object Code from Figure 3-2 into Memory (Default Allocation)

As Figure 3-4 shows, the linker:

1) Allocates the .text section first, beginning at address 0h. The .text section contains 10 words of object code.

2) Allocates the .data section next, beginning at address Ah. The .data section contains 7 words of object code.

3) Allocates the .bss section third, beginning at address 11h. The .bss section reserves 11 words in memory.

4) Allocates the named section `newvars` at address 1Ch. (`newvars` was the first named section encountered in the original input file – see Figure 3-2.) The `newvars` section reserves 8 words in memory.

5) Allocates the named section `vectors` at address 24h. The `vectors` section contains 2 words of object code.

Figure 3-5 shows a simple example of how *two* files might be linked together. When you link several files using the default algorithm, the linker combines all input sections that have the same name into one output section that has

this same name. For example, the linker combines the .text sections from two input files to create one .text output section.

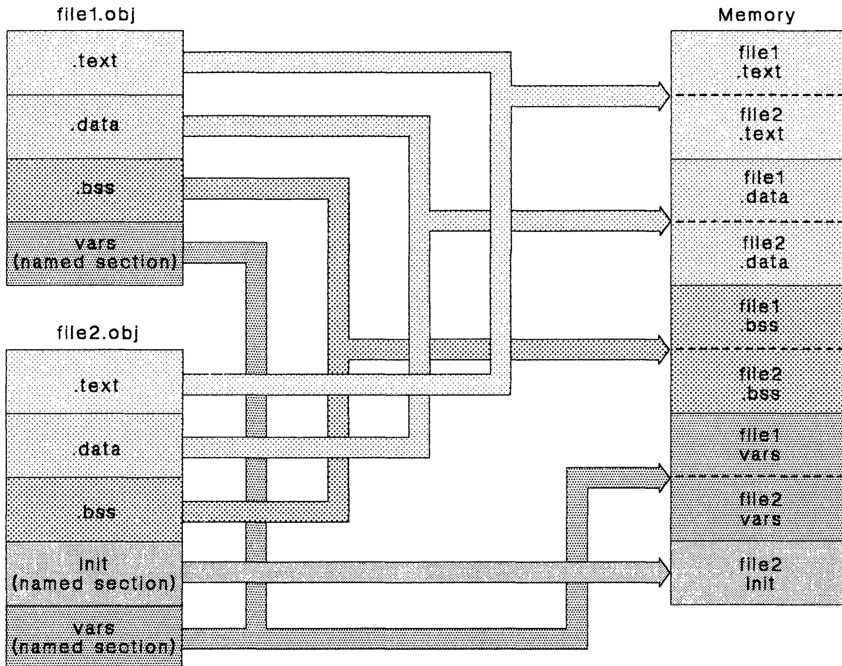


Figure 3-5. Combining Input Sections from Two Files (Default Allocation)

In Figure 3-5, `file1.obj` and `file2.obj` each contain the .text, .data, and .bss default sections and a named section called `vars`; `file2.obj` also contains a named section called `Init`. As Figure 3-5 shows, the linker:

1) Combines `file1` .text with `file2` .text to form one .text output section. The .text output section is allocated at address 0h.

2) Combines `file1` .data with `file2` .data to form the .data output section. The .data output section is allocated following the .text output section.

3) Combines `file1` .bss with `file2` .bss to form the .bss output section. The .bss output section is allocated following the .data output section.

4) Combines `file1` `vars` with `file2` `vars` to form the `vars` output section. (The `vars` section is the first named section that is encountered during the link, so it is allocated before the second named section, `Init`.) The `vars` output section is allocated following the .bss output section.

5) Allocates the `Init` section from `file2` after the `vars` section.

3.3.2 Placing Sections in the Memory Map

Figure 3-4 and Figure 3-5 illustrate the linker's default methods for combining sections and allocating them into memory. Sometimes you may not want to use the default setup. For example, you may not want to combine all of the .text sections into a single .text section. Or, you might want a named section placed at address 40h instead of the .text section. Most memory maps are comprised of various types of memories (DRAM, ROM, EPROM, etc.) in varying amounts; you may want to place a section in a particular type of memory.

The next two illustrations show another possible combination of the sections from Figure 3-4

- Figure 3-6 contains MEMORY and SECTIONS definitions.
- Figure 3-7 shows how the sections from figure Figure 3-6 are allocated into memory.

```
/**************************************************/
/*              Linker command file               */
/**************************************************/

MEMORY
{
   VECS:  origin = 000000h    length = 40h
   ROM:   origin = 000040h    length = FC0h
   RAM0:  origin = 801000h    length = 400h
   RAM1:  origin = 801400h    length = 400h
}

SECTIONS
{
   vectors   000000h     : { }
   .text                 : { }   > ROM
   .data                 : { }   > ROM
   .bss                  : { }   > RAM0
   newvars               : { }   > RAM1
}
```

Figure 3-6. MEMORY and SECTIONS Directives for Figure 3-7

- The MEMORY directive in Figure 3-6 defines four memory ranges:
  - VECS
  - ROM
  - RAM0
  - RAM1

  The *origin* for each of these ranges identifies the range's starting address in memory. The *length* specifies the length of the range. For example, memory range RAM0, with starting address 801000h and length 400h, defines the addresses 801000h through 8013FFh in memory.

- The SECTIONS directive in Figure 3-6 defines the order in which the sections are allocated into memory. The `vectors` section must begin at address 0. Both .text and .data are allocated into the `ROM` area that was defined by the MEMORY directive. The .bss section is allocated into `RAM0`, and `newvars` is allocated into `RAM1`.

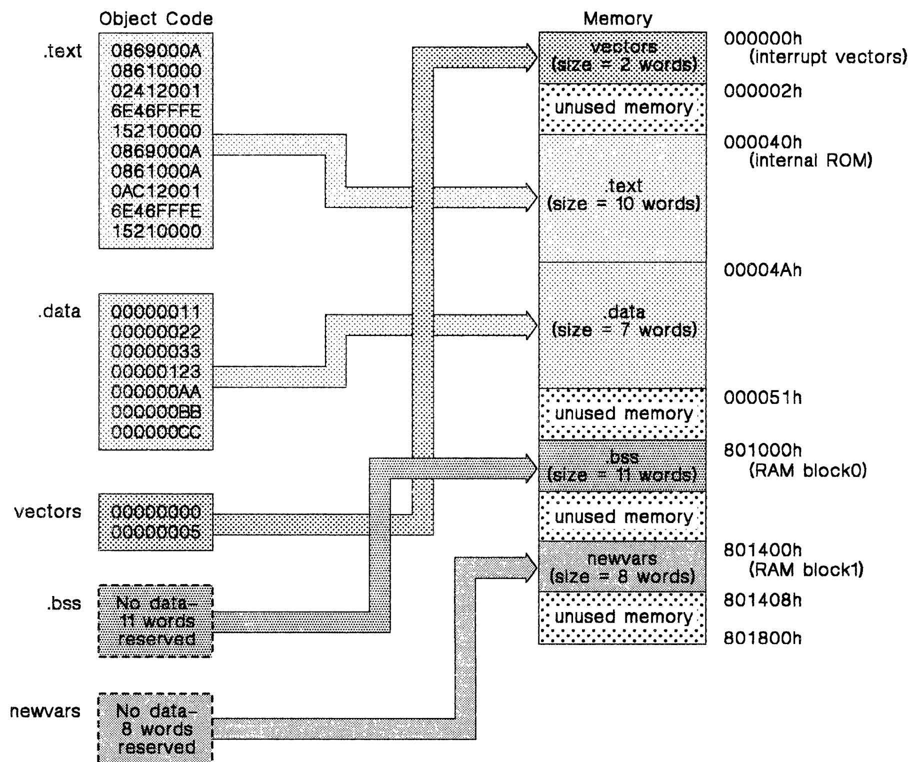


Figure 3-7. Rearranging the Memory Map from Figure 3-4

3.4 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 can't actually begin at address 0 in memory, so the linker **relocates** sections by:

- Allocating sections in the memory map so that they begin at the appropriate address,
- Adjusting symbol values to correspond to the new section addresses, **and**
- Patching 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. Figure 3-8 contains a code segment that generates relocation entries.

```
0001                            .ref    X
0002 00000000                   .text
0003 00000000 600000000!        BR      X       ; Generates a relocation entry
0004 00000001 082000002+        LDI     @Y,R0   ; Generates a relocation entry
0005 00000002 060000000  Y:     IDLE
```

Figure 3-8. An Example of Code that Generates Relocation Entries

In Figure 3-8, both the symbols X and Y are relocatable. X is defined in some other module; Y is defined in the .text section of this module. When assembled, X has a value of 0 (the assembler assumes all undefined external symbols have values of 0) and Y has a value of 2 (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 +).

After linking, suppose that X is relocated to address 100h. Suppose also that the .text section is relocated to begin at address 200h; Y now has a relocated value of 202h. The linker uses the two relocation entries to patch the two references in the object code:

```
60000000   BR   X        becomes     60000100
08200002   LDI  @Y,R0    becomes     08200202
```

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.

3.5 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; however, the sections in an executable object file are combined and relocated to fit into target memory.

In order 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. Some of the more common situations are listed below.

- The TMS320C30 debugging tools (including the software simulator, XDS emulator, and software development system) have built-in loaders. Each of these tools has a LOAD command that invokes a COFF loader; the loader reads the executable file and copies the program into target memory.

- If you are using a ROM- or EPROM-based system, you can use the object format converter (which is included with 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.

- Some TMS320C30 programs are loaded under the control of an operating system or monitor software running directly on the target system. In this type of application, the target system usually has an interface to the file system on which the executable module is stored. You must write a custom loader for this type of system. The loader must comprehend the file system (in order to access the file) as well as the memory organization of the target system (to load the program into memory).

3.6 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.

3.6.1 External Symbols

External symbols are symbols which are defined in one module and referenced in another module. You can use the .global directive to identify symbols as external. In a source module, an external symbol can be either:

- **Def**ined in the current module, **or**
- Defined in another module and **ref**erenced in the current module.

The following code segment illustrates these definitions.

```
x:      LDI       R0,R1      ; Define x
        LDI       @y,R0      ; Reference y
        .global   x          ; DEF of x
        .global   y          ; REF of y
```

The .global definition of x says that it is an external symbol defined in this module, and that other modules can reference x. The .global definition of y says that it is an undefined symbol that is defined in some other module.

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

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

3.6.2 The Symbol Table

The assembler always generates an entry in the symbol table when it encounters an external symbol (both definitions and references). 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 in a section.

The assembler does not usually create symbol table entries for any other type of symbol because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with .global. 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.

Section 4

Assembler Description

The assembler translates assembly language source files into machine language object files. These object files are in common object file format (COFF), discussed in Section 3. Source files can contain these assembly language elements:

- Assembler directives (described in Section 5),
- Assembly language instructions (summarized in Section 6), **and**
- Macro directives (described in Section 7).

The assembler:

- Is a two-pass assembler.
- 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.
- Appends a cross-reference listing to the source listing (if requested).
- Allows you to segment your code into sections.
- Maintains an **SPC** (section program counter) for each section of object code.
- Defines and references global symbols.
- Assembles conditional blocks.
- Supports macros, allowing you to define macros inline or in a macro library.

Section **Page**

4.1 Assembler Development Flow 4-2
4.2 Invoking the Assembler 4-3
4.3 Specifying Alternate Directories for Assembler Input 4-4
4.4 Source Statement Format 4-6
4.5 Constants 4-8
4.6 Character Strings 4-11
4.7 Symbols 4-11
4.8 Expressions 4-12
4.9 Source Listings 4-15
4.10 Cross-Reference Listings 4-17

4.1 Assembler Development Flow

Figure 4-1 illustrates the assembler's role in the assembly language development flow. The assembler accepts assembly language source files as input and creates a COFF object file that can be linked.

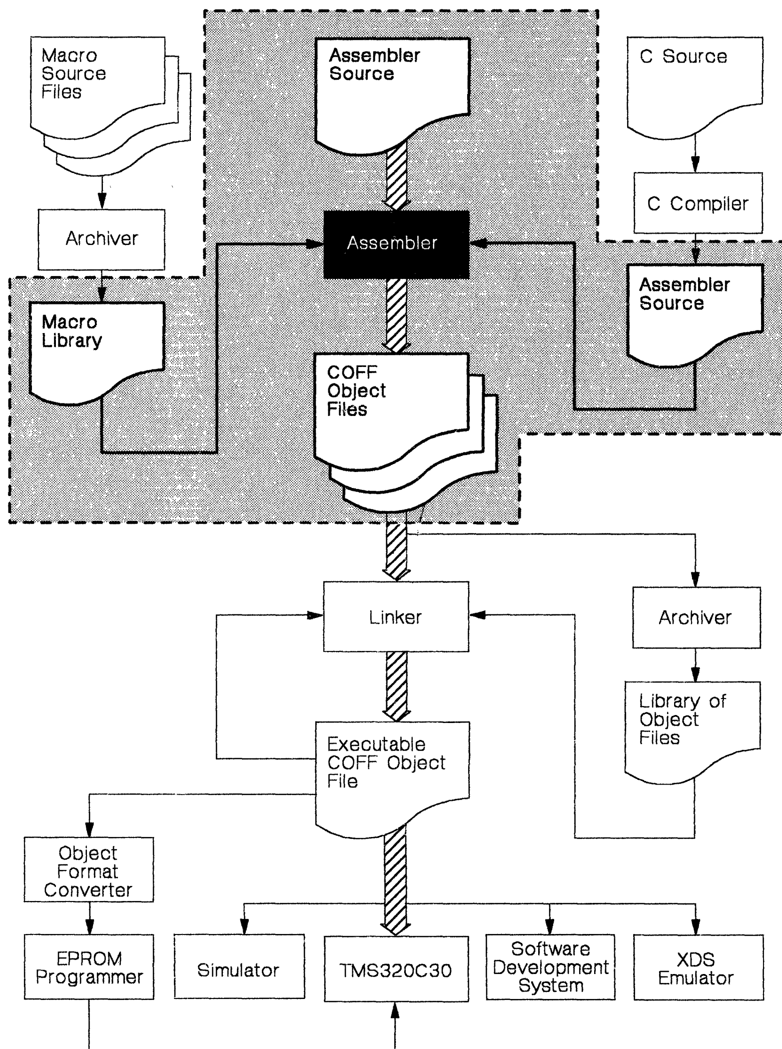


Figure 4-1. Assembler Development Flow

4.2 Invoking the Assembler

To invoke the assembler, enter:

asm30 *[input file [object file [listing file]]] [-options]*

input file names the assembler source file. If you do not supply an extension, the assembler assumes that the input file has the default extension ***.asm***. If you do not supply an input filename when you invoke the assembler, the assembler will prompt you for one.

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

listing file names the optional listing file that the assembler can create. If you do not supply a name for a listing file, *the assembler does not create one,* unless you use the -l (lowercase "L") option. In this case, the assembler uses the input filename with the *.lst* extension. If you supply a filename without an extension, the assembler uses *.lst.*

option identifies the assembler options that you want to use. *Case is insignificant for assembler options.* Options can appear anywhere on the command line; precede each option with a hyphen (-). You can string the options together; for example, **-lc** is equivalent to **-l -c**. Valid options include:

-l (lowercase "L") produces a listing file.

-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*. You can specify up to 10 directories in this manner; each pathname must be preceded by the -i option.

-x produces a cross-reference table and appends it to the end of the listing file. If you use -x but do not request a listing file, the assembler creates one anyway, but the listing contains only the cross-reference table.

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

-c makes case insignificant. For example, the symbols ABC and abc will be equivalent. *If you do not use this option, case is significant.*

-q (quiet) suppresses the banner and all progress information.

4.3 Specifying Alternate Directories for Assembler Input

The .copy and .include directives tell the assembler to read source statements from another file; the .mlib directive names a library that contains macro definitions. Section 5, Assembler Directives, provides examples of the .copy, .include, and .mlib directives. The syntax for these directives is:

.copy *"filename"*
.include *"filename"*
.mlib *"filename"*

The *filename* names a copy/include file that the assembler reads statements from or a macro library that contains macro definitions. The *filename* can be a complete pathname or a filename with no path information. If you provide a pathname, the assembler uses that path and *does not look* for the file in any other directories. If you do not provide path information, the assembler searches for the file in:

1) The directory that contains the current source file. (The current source file refers to the file that is being assembled when the .copy, .include, or .mlib directive is encountered.)

2) Any directories named with the -i assembler option.

3) Any directories set with the environment variable A_DIR.

You can augment the assembler's directory search algorithm by using the -i assembler option or the environment variables A_DIR.

4.3.1 -i Assembler Option

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

asm30 -i*pathname source filename*

You can use up to 10 -i options per invocation; each -i option names one *pathname*. In assembly source, you can now use the .copy, .include, or .mlib directive without specifying any path information. If the assembler doesn't find the file in the directory that contains the current source file, it searches the paths provided 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".
```

The complete path/filename for `copy.asm` is:

- `c:\c30\files\copy.asm` (DOS),
- `[c30.files]copy.asm` (VMS), **or**
- `/c30/files/copy.asm`(UNIX).

This is how you invoke the assembler:

DOS: `asm30 -ic:\c30\files source.asm`
VMS: `asm30 -i[c30.files] source.asm`
UNIX: `asm30 -i/c30/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.

4.3.2 Environment Variable (A_DIR)

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

DOS: **set** **A_DIR**=*pathname;another pathname ...*
VMS: **assign** **A_DIR** *"pathname;another pathname ... "*
UNIX: **setenv** **A_DIR** *"pathname;another pathname ... "*

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 now use the .copy, .include, or .mlib directive without specifying any path information. If the assembler doesn't find the file in the directory that contains the current source file *or* in directories named by -i, 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 that the complete path and file information for these copy files is:

- `c:\320\files\copy1.asm` and `c:\dsys\copy2.asm` (DOS),
- `[320.files]copy1.asm` and `[dsys]copy2.asm` (VMS), **or**
- `/320/files/cop1.asm` and `/dsys/cop2.asm` (UNIX)

This is how you set the environment variables and invoke the assembler:

DOS:
```
set A_DIR=c:\dsys; c:\exec\files
asm30 -ic:\320\files  source.asm
```

VMS:
```
assign  A_DIR "[dsys]; [exec.files]"
asm30  -i[320.files]        source.asm
```

UNIX:
```
setenv A_DIR "/exec/files;/dsys"
asm30 -i/320/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 icon -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:

DOS: `set     A_DIR=`
VMS: `deassign A_DIR`
UNIX: `setenv  A_DIR" "`

4.4 Source Statement Format

TMS320C30 assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. Source statement lines can be as long as the source file format allows. The assembler reads up to 200 characters per line. If the statement contains more than 200 characters, the assembler truncates the line and issues a warning.

The next several lines show examples of source statements:

```
SYM     .set  0A5h       ; Symbol SYM = 0A5h
Begin:  ADDI  SYM+5,R1   ; Add (SYM+5) to the contents of R1
        LDI   R1,R2      ; Move contents of R1 to R2
```

A source statement can contain four ordered fields. The general syntax for source statements is:

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

where

- 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. (Note that tab characters are equivalent to 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.

4.4.1 Label Field

Labels are optional for all assembly language instructions and for most (but not all) assembler directives. A label **must** begin in column 1 of a source statement. A label can contain up to 32 alphanumeric characters (A–Z, a–z, 0–9, _, and $). Labels are case sensitive, 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 don't use a label, then the first character position must contain a blank, a semicolon, or an asterisk.

When you use a label, its value is the current value of the section program counter (the label points to the statement it's associated with). If, for example, 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 3Fh.

```
 .      .        .                  .
 .      .        .                  .
 .      .        .                  .
0002                        * Assume some other code was assembled
0003 00003F 0000000A        Start:    .word    0Ah,3,7
     000040 00000003
     000041 00000007
```

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

```
label   .set    $       ; ($ represents 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):

```
0005 000042                 Here:
0006 000042 08010000                 LDI       R0,R1
```

4.4.2 Mnemonic Field

The mnemonic field follows the label field. *The mnemonic field cannot start in column 1, or it would be interpreted as a label.* The mnemonic field can contain one of the following opcodes:

- Machine-instruction mnemonic (such as ADDI, MPYF, LDI)
- Assembler directive (such as .data, .list, .set)
- Macro directive (such as $MACRO, $LOOP, $ENDLOOP)
- A macro invocation

4.4.3 Operand Field

The operand field is a list of operands that follows the mnemonic field. An operand can be a constant (see Section 4.5), a symbol (see Section 4.7), or a combination of constants and symbols in an expression (see Section 4.8). You must separate operands with commas.

4.4.4 Comment Field

A comment can begin in any column and extends to the end of the source line. A comment can contain any ASCII character including a blank. 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 ; or a *. Comments that begin anywhere else on the line **must** begin with a ;. The * symbol designates a comment only if it appears in column 1.

4.5 Constants

The assembler supports seven types of constants:

- Binary integer constants,
- Octal integer constants,
- Decimal integer constants,
- Hexadecimal integer constants,
- Floating-point constants,
- Character constants, **and**
- Assembly-time constants.

The assembler maintains each constant internally as a 32-bit quantity.

Note that constants **are not sign extended**. For example, the constant 0FFFFH is equal to $0000FFFF_{16}$ or $65{,}535_{10}$; it **does not** equal -1.

4.5.1 Binary Integers

A binary integer constant is a string of up to 32 binary digits (0s and 1s) followed by the suffix **B** (or **b**). If less than 32 digits are specified, the assembler right-justifies the value and zero-fills the unspecified bits. Examples of valid binary constants include:

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

4.5.2 Octal Integers

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

10Q	Constant equal to 8_{10} or 8_{16}
100000Q	Constant equal to $32{,}768_{10}$ or 8000_{16}
226Q	Constant equal to 150_{10} or 96_{16}

4.5.3 Decimal Integers

A decimal integer constant is a string of decimal digits ranging from -2,147,483,647 to 4,294,967,295. Examples of valid decimal constants include:

1000 Constant equal to 1000_{10} or $3E8_{16}$

-32768 Constant equal to $-32{,}768_{10}$ or -8000_{16}

25 Constant equal to 25_{10} or 19_{16}

4.5.4 Hexadecimal Integers

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

78h Constant equal to 120_{10} or 0078_{16}

0Fh Constant equal to 15_{10} or $000F_{16}$

37ACH Constant equal to $14{,}252_{10}$ or $37AC_{16}$

4.5.5 Character Constants

A character constant is a string of one to four characters enclosed in *single* quotes. The characters are represented internally as 8-bit ASCII characters. Two consecutive single quotes are required to represent each single quote within a character constant. A character constant consisting only of two single quotes (no letter) is valid and is assigned the value 0. If less than four characters are specified, the assembler right-justifies the bits. Examples of valid character constants include:

'ab' Represented internally as 00006261_{16}

'C' Represented internally as 00000043_{16}

'''D' Represented internally as 00004427_{16}

'abcd' Represented internally as 64636261_{16}

Note the difference between character *constants* and character *strings* (Section 4.6 discusses character strings). A character constant represents a single integer value; a string is a list of characters.

4.5.6 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 ] ]
```

where nnn is a string of decimal digits. A floating-point constant may be preceded with a + or a -. You must specify a decimal point; for example, 3.e5 is valid, but 3e5 is illegal. The exponent indicates a power of 10.

Valid floating-point constants include:

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

Floating-point constants **cannot be used in expressions**; the only valid floating-point operations are unary + and -. Floating-point constants that are used in instructions are represented in short format (16 bits). All other floating-point constants are represented in single-precision format (32 bits).

For more information about floating-point format, refer to the *Third-Generation TMS320 User's Guide.*

4.5.7 Assembly-Time Constants

If you use the .set directive to assign a constant value to a symbol, the symbol becomes an assembly-time constant. In order to use this constant in expressions, the value that is assigned to it must be absolute. For example:

```
sym    .set   3
       LDI    sym,R0     ; Load the constant 3 into R0
```

If you assign a floating-point constant to a symbol, then the symbol can be used only as a floating-point constant. Similarly, if you assign an integer constant to a symbol, then the symbol can be used only as an integer constant. The following example is *illegal*:

```
sym    .set   3          ; Integer constant
       LDF    sym,R0     ; Invalid - floating-point
                         ; constant required
```

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

```
sym    .set   R0
       LDI    10,sym
```

4.6 Character Strings

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

Examples of valid character strings include:

"sample program"	Defines a 14-character string, `sample program`
"PLAN ""C"""	Defines an 8-character string, `PLAN "C"`

Character strings are used for:

- Filenames (as in `.copy "filename"`)
- Section names (as in `.sect "section name"`)
- Data initialization directives (as in `.byte "charstring"`)

4.7 Symbols

Symbols are used as labels and in operands. A symbol name is a string of up to 32 alphanumeric characters (A–Z, a–z, 0–9, $, and _). The first character in a symbol cannot be a number; symbols cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler will recognize ABC, Abc, and abc as three unique symbols. (You can override this with the -c assembler option.) This type of symbol is valid only during the assembly in which it is defined, unless you use the .global directive to declare it as an external symbol.

Symbols that are used as **labels** become symbolic addresses that are associated with locations in the program. Labels **must** be unique; do not re-use them for other statements. Mnemonic opcodes and assembler directive names (without the "." prefix) are valid label names.

Symbols that are used in **operands** must be defined in the assembly by appearing as labels **or** as operands of a .global, .set, or .bss directive.

The assembler has several predefined symbols, including:

- $ (the dollar sign character), which represents the current value of the section program counter (SPC).

- These **register symbols**:

AR0–AR7	IF	IR0	RE	R0–R7
BK	IE	IR1	RC	SP
DP	IOF	PC	RS	ST

4.8 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. The range of valid expression values is -2,147,483,647 to 4,294,967,295.

Three main factors influence the order of expression evaluation:

- **Parentheses**. Expressions that are enclosed in parentheses are always evaluated first.

  Example: 8/(4/2) = 4, but 8/4/2 = 1

- **Precedence groups**. Operators (listed in Table 4-1) are divided into four precedence groups. When the order of expression evaluation is not determined by parentheses, the highest-precedence operation is evaluated first.

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

- **Left-to-right evaluation**. When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated from left to right. (Note that the highest-precedence group is evaluated from right to left.)

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

Note that all expressions are represented internally as 32-bit values. For example, -2 is represented as FFFF FFFEh, **not** as FFFEh.

4.8.1 Operators

Table 4-1 lists the operators that can be used in expressions. They are listed according to precedence group.

Table 4-1. Operators

<table>
<tr><th colspan="2">Group 1 (Highest Precedence)
Right-to-Left Evaluation</th><th colspan="2">Group 3
Left-to-Right Evaluation</th></tr>
<tr><td>+</td><td>Unary plus (positive expression)</td><td>+</td><td>Addition</td></tr>
<tr><td>-</td><td>Unary minus (negative expression)</td><td>-</td><td>Subtraction</td></tr>
<tr><td>~</td><td>(COM) 1s complement</td><td>|</td><td>(OR)Bitwise OR</td></tr>
<tr><td>!</td><td>(NOT) Logical NOT (if expr. = 0, 1 is returned, else 0 is returned)</td><td>^</td><td>(XOR) Bitwise exclusive OR</td></tr>
<tr><td></td><td></td><td>&</td><td>Bitwise AND</td></tr>
<tr><th colspan="2">Group 2
Left-to-Right Evaluation</th><th colspan="2">Group 4 (Relational Operators)
Left-to-Right Evaluation</th></tr>
<tr><td>*</td><td>Multiplication</td><td><</td><td>Less than</td></tr>
<tr><td>/</td><td>Division</td><td>></td><td>Greater than</td></tr>
<tr><td>%</td><td>(MOD) Modulo</td><td><=</td><td>Less than or equal to</td></tr>
<tr><td><<</td><td>(SHL) Shift left</td><td>>=</td><td>Greater than or equal to</td></tr>
<tr><td>>></td><td>(SHR) Shift right</td><td>=</td><td>Equal to (= =)</td></tr>
<tr><td></td><td></td><td><></td><td>Not equal to</td></tr>
</table>

Note: Operators in parentheses indicate an alternate form.

4.8.2 Expression Overflow or Underflow

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

4.8.3 Well-Defined Expressions

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

`1000h+X` Where `X` was previously defined as an absolute symbol.

4.8.4 Conditional Expressions

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

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

4.8.5 Relocatable Symbols and Legal Expressions

Table 4-2 summarizes valid operations on absolute, relocatable, and external symbols. An expression cannot multiply or divide by a relocatable or external symbol. An expression cannot contain unresolved symbols that are relocatable with respect to different sections.

Table 4-2. Expressions with Absolute and Relocatable Symbols

A is...	B is...	Results of A+B are...	Results of A-B are...
absolute	absolute	absolute	absolute
absolute	external	external	illegal
absolute	relocatable	relocatable	illegal
relocatable	absolute	relocatable	relocatable
relocatable	relocatable	illegal	absolute†
relocatable	external	illegal	illegal
external	absolute	external	external
external	relocatable	illegal	illegal
external	external	illegal	illegal

† A and B must be in the same section, otherwise this is illegal.

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

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

- **Example 1:**

  The statements in this example use an absolute symbol, `LAB1`. The first statement puts the value 51 into register AR0. The second statement loads the value 27 into register AR0.

  ```
  LDI     LAB1 + ((4+3) * 7), AR0   ; AR0 = 51
  LDI     LAB1 + 4 + 3 * 7, AR0     ; AR0 = 27
  ```

- **Example 2:**

  All legal expressions can be reduced to one of two forms:

  relocatable symbol ± absolute symbol

  or

  absolute value

  Unary operators can only be applied to absolute values; they cannot be applied to relocatable symbols. Expressions that cannot be reduced to contain only one relocatable symbol are illegal. The first statement in the following example is legal; the statements that follow it are not.

  ```
  LDI  extern_1 - 10, AR0              ; Legal
  LDI  10-extern_1, AR0                ; Can't negate reloc. symbol
  LDI  -(intern_1), AR0                ; Can't negate reloc. symbol
  LDI  extern_1/10, AR0                ; / isn't an additive operator
  LDI  intern_1 + extern_1, AR0        ; Multiple relocatables
  ```

- **Example 3:**

  The first statement below is legal; although `intern_1` and `intern_2` are relocatable, their difference is absolute because they're 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.

  ```
  LDI   intern_1 - intern_2 + extern_1,AR0    ; Legal
  LDI   intern_1 + intern_2 + extern_1,AR0    ; Illegal
  ```

- **Example 4:**

  An external symbol's placement in an expression is important to expression evaluation. Although the statement below is similar to the first statement in the previous example, it is illegal. This is because of left-to-right operator precedence; the assembler attempts to add `intern_1` to `extern_1`.

  ```
  LDI   intern_1 + extern_2 - intern_2, AR0    ; Illegal
  ```

4.9 Source Listings

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

At the top of each source listing page are two banner lines, a blank line, and a title line. Any title supplied by a .title directive is printed on this line; a page number is printed to the right of the title. If you don't use the .title directive, the title area is left blank. The assembler inserts a blank line below the title line.

Each line in the source file produces a line in the listing file that contains a source statement number, an SPC value, the object code assembled, and the source statement. A source statement may produce more than one word of object code. The assembler lists the SPC value and object code on a separate line for each additional word. Each additional line is listed immediately following the source statement line.

```
1    2       3        4
0027 000006  03E20018 Begin: ASH   24,R2   ; shift to top of word
0028 000007  02000002        ADDI  R2,R0   ; add to LSBs
0029 000008  78800000        RETS
```

Field 1 *Source Statement Number.* The source statement number is a 4-digit 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 statements in the source file that are not listed. Source lines generated by a macro call, a .copy directive, or an .include directive are renumbered starting at 0001. The original sequence continues after the copying or macro expansion is complete. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. An **A** indicates the first level, **B** indicates a second level, etc.

Field 2 *Section Program Counter.* This field contains the section program counter, or **SPC**, value (hexadecimal). Each section (.text, .data, .bss, and named sections) maintains a separate SPC. Some directives do not affect the SPC; they 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 by appending one of the following characters to the end of the field:

! Undefined external reference
' Relocatable with respect to the .text section
" Data relocatable (.data, .sect)
\+ .bss relocatable

Field 4 *Source Statement Field.* This field contains the characters of the source statement as they were scanned by the assembler. The maximum line length accepted by the assembler is 200 characters. Spacing in this field is determined by the spacing in the source statement.

```
TMS320C30 Assembler      Version 1.0 87.100        Fri May 29 14:13:54 1987
(c) Copyright 1987, Texas Instruments Inc.
TMS320C30 Integer Multiply                                          PAGE    1

0002                        *******************************************************
0003                        * TMS320C30 32x32 Integer Multiply
0004                        *
0005                        *   Inputs: x in R0, y in R1
0006                        *           AR0 points to 2 words of temporary memory
0007                        *
0008                        *   Outputs: x * y in R0
0009                        *
0010                        *   Operation:
0011                        *        Let x0 = 8 MSBs of x, y0 = 8 MSBs of y
0012                        *
0013                        *        result = (x0 * y) + (y0 * x) + xy
0014                        *******************************************************
0015
0016                              .global mpy32
0017
0018 000000                 mpy32:
0019 000000 C20100C0              STI     R0,*AR0          ; save x
0020                        ||    STI     R1,*+AR0         ; save y
0021 000001 03E0FFE8              ASH     -24,R0           ; x0 into R0
0022 000002 03E1FFE8              ASH     -24,R1           ; y0 into R1
0023 000003 0AC00001              MPYI    *+AR0,R0         ; mpy upper bytes: x0 * y
0024 000004 0AC1C000              MPYI    *AR0,R1          ;                  y0 * x
0025 000005 880800C0              MPYI    *AR0,*+AR0,R0 ; mpy lower words
0026                        ||    ADDI    R0,R1,R2         ; add product MSBs
0027 000006 03E20018              ASH     24,R2            ; shift back to top of word
0028 000007 02000002              ADDI    R2,R0            ; add to LSBs
0029 000008 78800000              RETS
0030                              .end

No Errors,   No Warnings
```

Figure 4-2. Sample Assembler Listing

4.10 Cross-Reference Listings

A cross-reference listing shows symbols and their definitions. To obtain a cross-reference listing, invoke the assembler with the -x option or use the .option directive. The assembler will append the cross-reference to the end of the source listing.

```
TMS320C30 Assembler      Version 1.0 87.100          Fri May 29 14:13:54 1987
(c) Copyright 1987, Texas Instruments Inc.
                                                                   PAGE    3

LABEL                       VALUE       DEFN    REF

K16                         0000AABB    0007    0041
K24                         00AABBCC    0008    0049
K32                         AABBCCDD    0009    0057
K8                          000000AA    0006    0071
KFLOAT                      E5541885    0010    0065
ext                         REF                 0011  0026  0034  0042  0050
                                                0058  0072
label0                      00000002+   0019    0028  0036  0044  0052  0060
                                                0074
label1                      00000003'   0028    0027  0035  0043  0051  0059
                                                0073
```

Figure 4-3. Cross-Reference Listing Format

- The **label** column contains each symbol that was defined or referenced during the assembly.
- The **value** column contains a 4-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 followed by a character that describes the symbol's attributes. Table 4-3 lists these characters and names.
- The **definition** (DEFN) column contains the statement number that defines the symbol. This column is blank for undefined symbols.
- The **reference** (REF) column lists the line numbers of statements that reference the symbol. A blank in this column indicates that the symbol was never used.

Table 4-3. Symbol Attributes for Cross-Reference Listings

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 .bss section

Section 5

Assembler Directives

Assembler directives supply program data and control the assembly process. Assembler directives allow you to:

- Reserve space in memory for uninitialized variables
- Control the appearance of listings
- Initialize memory
- Assemble conditional blocks
- Define global variables
- Specify libraries that the assembler can obtain macros from
- Examine symbolic debugging information

This section is divided into two parts: the first part (Sections 5-1 through 5-7) describes the directives according to function, and the second part (Section 5.8) is an alphabetical reference. You will find the following topics in this section:

Section **Page**

5.1 Directives Summary 5-2
5.2 Sections Directives 5-4
5.3 Directives that Initialize Memory 5-6
5.4 Directives that Align the Section Program Counter 5-9
5.5 Directives that Format the Output Listing 5-10
5.6 Conditional Assembly Directives 5-11
5.7 Directives that Reference Other Files 5-12
5.8 Directives Reference 5-13

The TMS320C30 C compiler uses several directives for symbolic debugging. Unlike other directives, symbolic debugging directives are not used in most assembly language programs. Appendix B discusses these directives; they are not discussed in this section.

5.1 Directives Summary

Table 5-1 summarizes the assembler directives. *Note that all source statements that contain a directive may have a label and a comment.* To improve readability, they are not shown as part of the directives' syntax.

Table 5-1. Directives Summary

Sections Directives	
Mnemonic and Syntax	**Description**
.asect *"section name", address*	Assemble into an absolute named (initialized) section
.bss *symbol, size in words*	Reserve *size* words in the .bss (uninitialized data) section
.data	Assemble into the .data (initialized data) section
.label *"symbol"*	Define a label in an absolute section
.sect *"section name"*	Assemble into a named (initialized) section
.text	Assemble into the .text (executable code) section
symbol .usect *"section name", size in words*	Reserve *size* words in a named (uninitialized) section
Directives that Initialize Memory	
Mnemonic and Syntax	**Description**
.byte *value*$_1$ *[,..., value*$_n$*]*	Initialize one or more successive bytes in the current section
.field *value [,size in bits]*	Initialize a variable-length field
.float *value*$_1$ *[,..., value*$_n$*]*	Initialize one or more 32-bit, single-precision, floating-point constants
.hword *value*$_1$ *[,..., value*$_n$*]*	Initialize one or more 16-bit (half-word) values
.int *value*$_1$ *[,..., value*$_n$*]*	Initialize one or more 32-bit integers
.long *value*$_1$ *[,..., value*$_n$*]*	Initialize one or more 32-bit integers
symbol .set *value*	Initialize an assembly-time constant
.space *size in words*	Reserve *size* words in the current section
.string *"string*$_1$*" [,..., "string*$_n$*"]*	Initialize one or more text strings
.word *value*$_1$ *[,..., value*$_n$*]*	Initialize one or more 32-bit integers
Directives that Align the Section Program Counter (SPC)	
Mnemonic and Syntax	**Description**
.align	Align the SPC on a 32-word (cache) boundary
.even	Align the SPC on a word boundary
Directives that Format the Output Listing	
Mnemonic and Syntax	**Description**
.length *page length*	Set the page length of the source listing
.list	Restart the source listing
.mlist	Allow macro listings (default)
.mnolist	Inhibit macro listings
.nolist	Stop the source listing

Table 5-1. Directives Summary (Concluded)

Directives that Format the Output Listing (continued)	
Mnemonic and Syntax	**Description**
.option {B|D|F|L|M|T|X}	Select output listing options
.page	Eject a page in the source listing
.title *"string"*	Print a title in source page heading
.width *page width*	Set the page width of the source listing
Conditional Assembly Directives	
Mnemonic and Syntax	**Description**
.if *expression*	Begin conditional assembly
.else	Optional conditional assembly
.endif	End conditional assembly
Directives that Reference Other Files	
Mnemonic and Syntax	**Description**
.copy *["]filename["]*	Include source statements from another file
.def *$symbol_1$ [,..., $symbol_n$]*	Identify one or more symbols that are defined in the current module and used in other modules
.global *$symbol_1$ [,..., $symbol_n$]*	Identify one or more global (external) symbols
.include *["]filename["]*	Include source statements from another file
.mlib *["]filename["]*	Specify the name of a macro library
.ref *$symbol_1$ [,..., $symbol_n$]*	Identify one or more symbols that are used in the current module but defined in another module
Miscellaneous Directives	
Mnemonic and Syntax	**Description**
.end	Program end
Symbolic Debugging Directives†	
Mnemonic and Syntax	**Description**
.block *beginning line number*	Begin a C block
.endblock *ending line number*	End a C block
.endfunc *ending line number*	End a function definition
.eos	End a structure, enumeration, or union definition
.etag *name,size*	Begin an enumeration definition
.file *"filename"*	Define a program identifier
.func *beginning line number*	Begin a function definition
.line *line number [,address]*	Specify the line number of a C source statement
.member *name, value [,type, storage class, size, tag, dims]*	Define a member of a structure, enumeration, or union
.stag *name, size*	Begin a structure definition
.sym *name, value [,type, storage class, size, tag, dims]*	Specify symbolic debug information for a global variable, local variable, or a function
.utag *name, size*	Begin a union definition

† Symbolic debugging directives are discussed in Appendix B

5.2 Sections Directives

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

- The **.bss** directive reserves space in the .bss section for variables.
- 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.
- The **.text** directive identifies portions of code in the .text section. The .text section usually contains executable code.
- The **.data** directive identifies portions of code in the .data section. The .data section usually contains initialized data.
- The **.sect** directive defines initialized named sections, and associates subsequent code or data with that section. Named sections are initialized and contain code or data.
- The **.asect** directive creates initialized named sections that have *absolute addresses.* (Within an absolute section, you can use the **.label** directive to define labels with absolute addresses.)

  Section 3 discusses COFF sections in detail.

Figure 5-1shows 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 section program counter. 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, its SPC will resume counting as if there had been no intervening code.

After the code in Figure 5-1 is assembled, the sections contain the following:

.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

Note that 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.

```
0001                               ********************************************
0002                               *  Start assembling into the .text section  *
0003                               ********************************************
0004 000000                                  .text
0005 000000 00000001                         .word     1, 2
     000001 00000002
0006 000002 00000003                         .word     3, 4
     000003 00000004
0007
0008                               ********************************************
0009                               *  Start assembling into the .data section  *
0010                               ********************************************
0011 000000                                  .data
0012 000000 00000009                         .word     9, 10
     000001 0000000A
0013 000002 0000000B                         .word     11, 12
     000003 0000000C
0014
0015                               ********************************************
0016                               *  Start assembling into named section,     *
0017                               *  var_defs                                 *
0018                               ********************************************
0019 000000                                  .sect     "var_defs"
0020 000000 00000011                         .word     17, 18
     000001 00000012
0021
0022                               ********************************************
0023                               *  Resume assembling into the .data section *
0024                               ********************************************
0025 000004                                  .data
0026 000004 0000000D                         .word     13, 14
     000005 0000000E
0027
0028 000000                                  .bss      sym,19      ; Reserve space in .bss
0029
0030 000006 0000000F                         .word     15, 16      ; Still in .data
     000007 00000010
0031
0032                               ********************************************
0033                               *  Resume assembling into the .text section *
0034                               ********************************************
0035 000004                                  .text
0036 000004 00000005                         .word     5, 6
     000005 00000006
0037
0038 000000                        usym      .usect    "xy",20      ; Reserve space in xy
0039
0040 000006 00000007                         .word     7, 8        ; Still in .text
     000007 00000008
0041
```

Figure 5-1. Examples of Sections Directives

5.3 Directives that Initialize Memory

Several directives assemble values into the current section:

- The **.set** directive equates a value with a symbol. This type of symbol is known as an *assembly-time constant*; it can be used in the same manner as a numeric constant (for example, in expressions).

  This example defines a symbol named `bval` and assigns the value 4 to it. The symbol `bval` can then be used as a constant.

  ```
  0001        00000004   bval      .set      4
  0002 000000 00000004             .byte     bval, bval*2, bval+12
       000001 00000008
       000002 00000010
  ```

  Note that the set directive produces no object code.

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

- The **.hword** directive places one or more 16-bit half-word values into consecutive words in the current section. This directive is similar to .word, except that the width of each value is restricted to 16 bits.

- The **.word**, **.int**, and **.long** directives place one or more 32-bit values into consecutive locations in the current section.

- The **.string** directive places 8-bit characters from one or more character strings into the current section. This directive is similar to .byte, except that four 8-bit values are packed into each word. The last word in a string is padded with null characters (0s) if necessary.

- The **.float** directive calculates the single-precision (32-bit) floating-point representations of specified floating-point values, and stores them in consecutive words in the current section. Here's an example of a .float directive and the object code that it generates:

  ```
  0005 000003 0274ED91            .float    7.654
  ```

Figure 5-2 compares the .byte, .hword, .word, and .string directives; for this example, assume the following code was assembled:

```
0001 000000 000000AB          .byte      0ABh
0002 000001 0000CDEF          .hword     0CDEFh
0003 000002 89ABCDEF          .word      089ABCDEFh
0004 000003 706C6568          .string    "help"
```

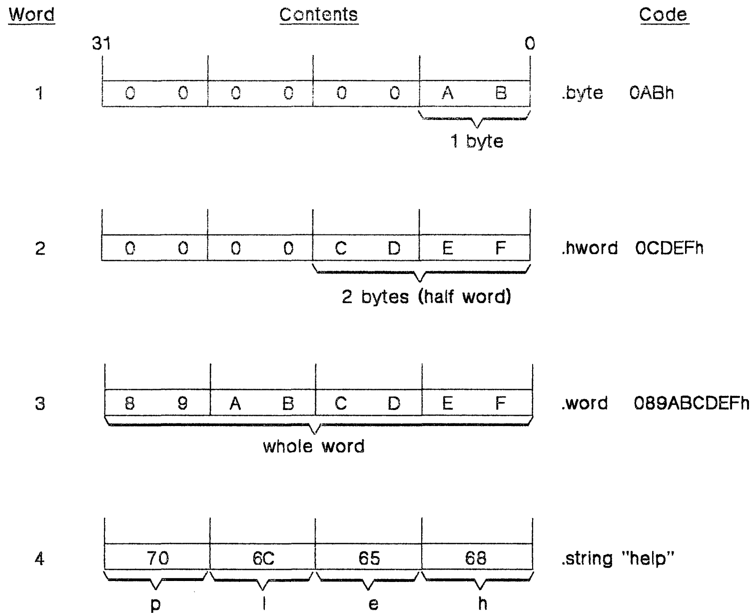


Figure 5-2. Examples of Initialization Directives

- The **.field** directive places a single value into a specified number of bits in the current word. You can pack multiple fields into a single word; the assembler will not increment the SPC until a word is filled.

  Figure 5-3 shows how fields are packed into a word. For this example, assume the following code has been assembled; notice that the SPC doesn't change (the fields are packed into the same word):

```
0006 000004 00000003          .field    3,4
0007 000004 00000083          .field    8,5
0008 000004 00002083          .field    16,7
```

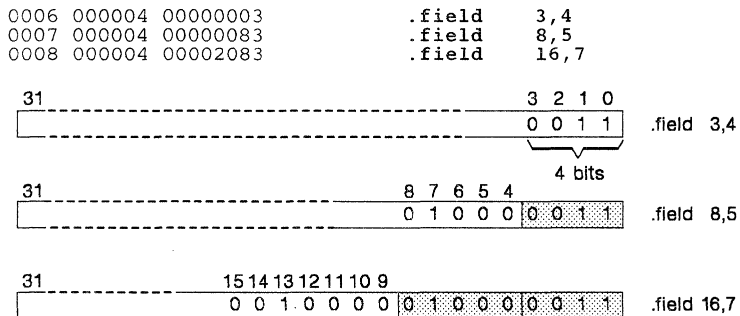


Figure 5-3. An Example of the .field Directive

- The **.space** directive reserves a specified number of words in the current section. The assembler fills these reserved words with 0s.

  Figure 5-4 shows an example of the .space directive; assume the following code has been assembled:

```
                .
                .
                .
0154 00027A 080F000C          LDI    AR4,AR7
0155 00027B 00000000          .space 27
0156 000296 0000000F          .word  15
```

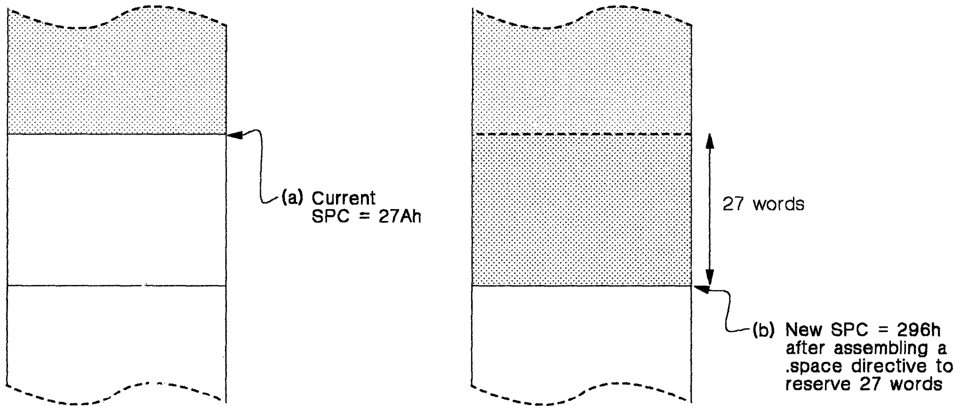


Figure 5-4. An Example of the .space Directive

5.4 Directives that Align the Section Program Counter

- The **.align** directive aligns the SPC on a 32-word boundary. This ensures that the code following the .align directive begins on a cache boundary. If the SPC is already aligned at a 32-word boundary, then it is not incremented and .align has no effect. Figure 5-5 shows an example of the .align directive; assume that the following code has been assembled:

```
0201 000C11 00000000        .align
0202 000C20 00000004        .byte  4
```

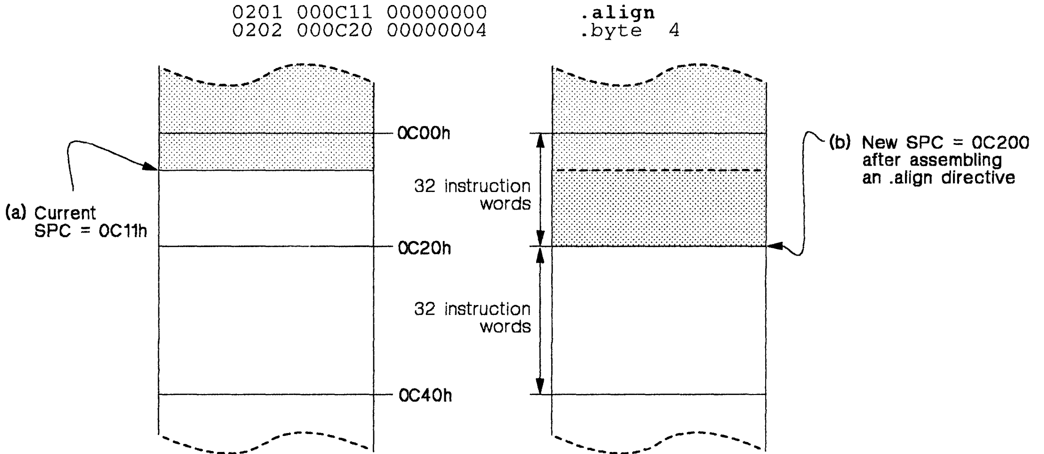


Figure 5-5. An Example of the .align Directive

- The **.even** directive aligns the SPC so that it points to the next full word. You should use .even after using .field directives; if the .field directive doesn't fill a word, the .even directive forces the assembler to write out the full word and fill the unused bits with 0s.

  Figure 5-3 (page 5-7) illustrates the .field directive; Figure 5-6 shows the effect of assembling a .even directive after a .field directive. Assume the following code has been assembled:

```
0006 000004 00000003          .field    3,4
0007 000004 00000083          .field    8,5
0008 000004 00002083          .field    16,7
0009 000005                   .even
```

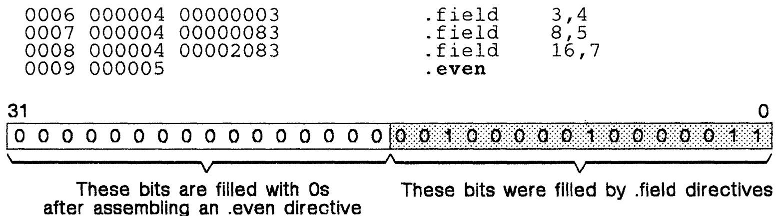


Figure 5-6. An Example of the .even Directive

5.5 Directives that Format the Output Listing

Seven directives format the listing file:

- The **.length** directive controls the page length of the listing file. You can use this directive to adjust listings for various output devices.
- The **.width** directive controls the page width 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 stop the assembler from printing selected source statements in the listing file. Use the .list directive to turn the listing back on.
- The **.mlist** and **.mnolist** directives allow and inhibit macro expansion listings.
- The **.option** directive controls several features in the listing file. This directive has several operands:

  B Limits the listing of .byte directives to 1 line.
  H Limits the listing of .hword directives to 1 line.
  F Resets the B, H, L, M, and T options.
  L Limits the listing of .long, .int, and .word directives to 1 line.
  M Limits macro expansions to 1 line.
  T Limits the listing of .string directives to 1 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.)

- The **.page** directive causes a page eject in the output listing.
- The **.title** directive supplies a title that the assembler prints on the second line of each page.

5.6 Conditional Assembly Directives

Three directives allow you to assemble conditional blocks of code:

- The **.if** directive marks the beginning of a conditional block. The .if directive has one parameter, which is an expression.
  - If this expression evaluates to *true* (a nonzero value), then the assembler assembles the code that follows it (up to an .else or .endif).
  - If this expression evaluates to *false* (0), then the assembler assembles code that follows an .else (if present) or an .endif (if no .else is present).
- The **.else** directive identifies a block of code that the assembler assembles if the if-expression is false (0). This directive is optional in the conditional block; if an expression is false and there is no .else statement, then the assembler continues with the code that follows the .endif.
- The **.endif** directive terminates a conditional block.

The assembler supports several relational operators that are especially useful for conditional expressions; see Section 4.8.4 on page 4-13 for more information about relational operators. Figure 5-7 shows an example of conditional assembly.

```
0001          00000001   sym1     .set     1
0002          00000002   sym2     .set     2
0003          00000003   sym3     .set     3
0004          00000004   sym4     .set     4
0005                     If_1:    .if      sym1 < sym2
0006 000000   00000001            .byte    sym1
0007                              .else
0008                              .byte    sym2
0009                              .endif
0010                     If_2:    .if      sym1 + sym2 = sym4
0011                              .byte    sym1 + sym2
0012                              .else
0013 000001   00000004            .byte    sym4
0014                              .endif
0015                     If_3:    .if      sym1 <> sym4 - sym2
0016 000002   00000001            .byte    sym1
0017                              .else
0018                              .byte    sym4 - sym2
0019                              .endif
```

Figure 5-7. An Example of Conditional Assembly Directives

5.7 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 is done 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 **.global** directive declares a symbol to be external so that it is available to other modules at link time. The .global directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. Note that the linker will resolve an undefined global symbol only if it is used in the program.
- The **.def** directive identifies a symbol that is defined in the current module and can be used by other modules. The assembler puts the symbol in the symbol table.
- 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 puts it in the object symbol table so the linker can resolve its definition.
- 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 will then be able to search for it in the specified macro library.

5.8 Directives Reference

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

Directive	Page
.align	5-14
.asect	5-15
.bss	5-17
.byte	5-18
.copy	5-19
.data	5-20
.def	5-26
.else	5-29
.end	5-21
.endif	5-29
.even	5-22
.field	5-23
.float	5-25
.global	5-26
.hword	5-28
.if	5-29
.include	5-18
.int	5-30
.label	5-15
.length	5-31
.list	5-32
.long	5-30
.mlib	5-33
.mlist	5-34
.mnolist	5-34
.nolist	5-32
.option	5-35
.page	5-36
.ref	5-26
.sect	5-37
.set	5-38
.space	5-39
.string	5-40
.text	5-41
.title	5-42
.usect	5-43
.width	5-31
.word	5-30

Syntax **.align**

Description The .align directive aligns the section program counter on the next 32-word boundary. If necessary, the assembler assembles words containing NOPs. This directive is useful for aligning code on a cache boundary.

Using the .align directive has two effects:

- The assembler aligns the SPC on a 32-word boundary *within* the current section.
- The assembler sets a flag that forces the linker to align the entire section on a 32-word boundary. This ensures that individual alignments remain intact when a section is loaded into memory.

Example This example aligns the SPC on the next 32-word boundary to ensure that the code that follows it will start on a cache boundary. Figure 5-8 shows how this code aligns the SPC.

```
0001 000000 08010000              LDI       R0,R1
0002
0003 000020                       .align
0004
0005 000020 08010000    x:        LDI        R0,R1
0006 000021 08010000              LDI        R0,R1
0007 000022 00000000              .space     25
0008
0009 000040                       .align
```

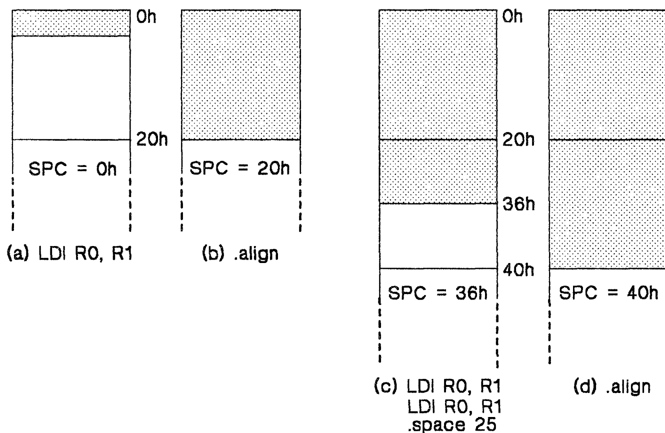


Figure 5-8. An Example of the .align Directive

Syntax

.asect *"section name" [, address]*

.label *symbol*

Description

The .asect directive defines a named section whose addresses are absolute with respect to *address*.

- The *section name* is a required parameter that identifies the name of the absolute section. The *name* must be enclosed in double quotes.
- The *address* required parameter identifies the section's absolute starting address in target memory. This address is **required** the first time that you assemble into a specific absolute section. If you use .asect to continue assembling into an absolute section that already contains code, you **cannot** use the address parameter.

Absolute sections are useful for loading sections of code from off-chip memory into faster on-chip memory. In order to use an absolute section, you must know which location you want the section to execute from, and specify it as the *address* parameter.

Most sections directives create sections with relocatable addresses. The starting SPC value for these sections is always zero; the linker then relocates them where appropriate. The starting SPC value for an absolute section, however, is the specified *address*. The addresses of all code assembled into an absolute section are offsets from the specified address. The linker *does* relocate sections defined with .asect; however, any labels defined within an absolute section retain their absolute (*runtime*) addresses. Thus, references to these labels refer to their *runtime* addresses, even though the section is not initially loaded at its runtime address.

All labels in an absolute section have absolute addresses. The .label directive creates "labels" with relocatable addresses; this allows you to define a symbol that points to the section's loadtime location in off-chip memory. The .label directive can only be used within an absolute section.

Note that after you define a section with .asect, you can use the .sect directive later in the program to continue assembling code into the absolute section.

Example

This example defines an absolute section called `abs`. At run time, this section will start at address 100h in on-chip RAM. `copy_start` is a relocatable symbol that points to the section's loadtime address in off-chip ROM. The symbols `abs_code` and `abs_end` are absolute addresses; `abs_code - abs_end` yields the number of lines of code to move. The function `copy` copies the section from ROM into RAM.

Figure 5-9 shows how the code is copied from one part of memory to another.

```
0001                               ***********************************************
0002                               * Define an absolute section. This section can *
0003                               * be linked and loaded into external ROM, then *
0004                               * copied into internal RAM and run.            *
0005                               ***********************************************
0006 000100                                  .asect    "abs", 100h   ; dest addr in RAM
0007 000100                                  .label    copy_start    ; src addr in ROM
0008 000100 08600000    abs_code:    LDI       0,R0
0009 000101 13FB0064                 RPTS      100
0010 000102 02402001                 ADDI      *AR0++,R0
0011 000103 78800000    abs_end:     RETS
0012
0013                               ***********************************************
0014                               * This function copies the absolute section    *
0015                               * from ROM to RAM.                             *
0016                               ***********************************************
0017 000000                          .data
0018 000000 00000100+   L1           .word     copy_start     ; ptr to code in ROM
0019 000001 00000100+   L2           .word     abs_code       ; dest addr in RAM
0020
0021 000000                          .text
0022 000000 08280000+   copy:        LDI       @L1,AR0        ; load src ptr
0023 000001 08290001+                LDI       @L2,AR1        ; load dest ptr
0024 000002 08402001                 LDI       *AR0++,R0      ; load first word
0025 000003 13FB0003                 RPTS      abs_end - abs_code
0026 000004 DA002120                 LDI       *AR0++,R0      ; copy all bytes
0027                              || STI       R0,*AR1++
0028 000005 78800000                 RETS                     ; end copy
0029
0030                               ***********************************************
0031                               * Main program -- copy the routine into RAM,   *
0032                               * then run.                                    *
0033                               ***********************************************
0034 000006 62000000+   run:         CALL      copy
0035 000007 62000100                 CALL      abs_code
0036 000008 78800000                 RETS
```

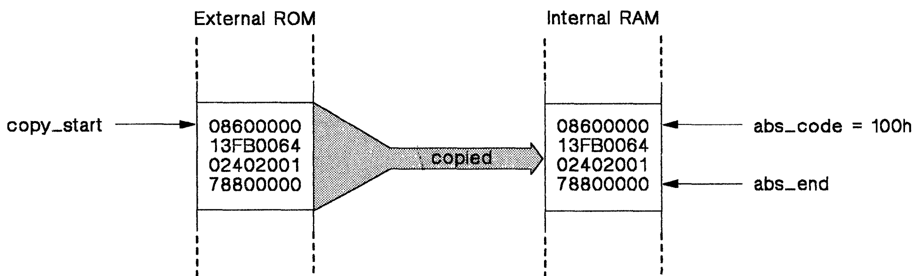


Figure 5-9. An Example of the .asect Directive

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

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

- The *symbol* is a required parameter. It defines a symbol that points to the first location reserved by the directive. The symbol name should correspond to the variable that you're reserving space for.
- The *size* is a required parameter; it must be an absolute expression. The assembler allocates *size* words in the .bss section. There is no default size.

Note that the .usect directive is similar to the .bss directive; it also reserves space in memory. However, .usect creates *named* uninitialized sections that can be allocated separately from the .bss section.

Other section directives (.text, .data, .sect, and .asect) 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 Section 3.

Example This example uses the .bss directive to allocate space for two variables, `array` and `dflag`. The symbol `array` points to 100 words of uninitialized space (the .bss SPC = 0). The symbol `dflag` points to 1 word of uninitialized space (the .bss SPC = 100). This example reserves a total of 101 words in the .bss section. Note that symbols declared with the .bss directive can be referenced in the same manner as other symbols and can also be declared external.

```
0001                        ********************************
0002                        * Begin assembling into .text  *
0003                        ********************************
0004 000000                       .text
0005 000000 08010000              LDI    R0,R1
0006
0007                        ********************************
0008                        *  Allocate 100 words in .bss  *
0009                        ********************************
0010 000000                       .bss   array,100
0011
0012                        ********************************
0013                        *       Still in .text         *
0014                        ********************************
0015 000001 08020001              LDI     R1,R2
0016
0017                        ********************************
0018                        *   Allocate 1 word in .bss    *
0019                        ********************************
0020 000064                       .bss    dflag,1
0021
0022                        ********************************
0023                        *       Still in .text         *
0024                        ********************************
0025 000002 08020064+             LDI     @dflag,R0
0026
0027                        ********************************
0028                        * Declare external .bss symbol *
0029                        ********************************
0030                              .global array
```

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

Description The .byte directive places one or more 8-bit values into consecutive words in the current section. Each *value* can be either:

- An expression which the assembler evaluates and treats as an 8-bit signed number.
- A character string enclosed in double quotes. Each character represents a separate value.

Values are not packed or sign extended; each byte value occupies the least significant 8 bits of a full 32-bit word. The assembler truncates values that are greater than 8 bits. You can use up to 100 values, but the total line length cannot exceed 200 characters. Each character in a string is counted as a separate operand.

If you use a label, it points to the location at which the assembler places the first byte.

Example This example places the 8-bit values 10, -1, 97, 98, 99, and 97 into six consecutive words in memory. The label `strx` has the value 64h, which is the location of the first initialized word.

```
0002 000064 0000000A  strx:    .byte    10,-1,"abc",'a'
     000065 000000FF
     000066 00000061
     000067 00000062
     000068 00000063
     000069 00000061
```

Syntax

.copy *["]filename["]*

.include *["]filename["]*

(The quote marks surrounding the filename are optional.)

Description

The .copy and .include directives tell the assembler to read source statements from a different file. The assembler:

1) Stops assembling statements in the current source file.

2) Assembles the statements in the copied/included file, **and**

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; the *filename* may be enclosed in double quotes. The *filename* must follow operating system conventions. You can specify a full pathname (for example, `.copy c:\dsp\file1.asm`). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file.

2) Any directories named with the -i assembler option.

3) Any directories specified by the environment variable A_DIR.

For more information about the -i option and the environment variable, see Section 4.3, Specifying Alternate Directories for Assembler Input, on page 4-4.

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 that are assembled.

The .copy and .include directives may be nested within a file being copied or included. The assembler limits this type of nesting to eight 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 This example uses the .copy directive to read and assemble source statements from other files and then resumes assembling into the current file.

Listing file:

```
 0001 000000 00000000              .space      20
 0002                              .copy       "byte.asm"
A0001                           ** In byte.asm
A0002 000014 00000020              .byte       32, 1+'A'
      000015 00000042
A0003                              .copy       "word.asm"
B0001                           ** In word.asm
B0002 000016 0000AABB              .word       0AABBh, 56q
      000017 0000002E
A0004                           ** Back in byte.asm
A0005 000018 0000006A              .byte       67h+3
 0003
 0004                           ** Back in original file
 0005 000019 656E6F44              .string     "Done"
```

Example 2 This example uses the .include directive to read and assemble source statements from other files and then resumes assembling into the current file.

include.asm (source file)	byte2.asm (first include file)	word2.asm (second include file)

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

Listing file

```
0001 0000            .space   29
0002                 .include "byte2.asm"
```

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.

Section 3 provides a detailed explanation about COFF sections.

Note that 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 specify an explicit section control directive.

Example This example assembles code into the .data and .text sections.

```
0001                            ****************************************
0002                            **      Reserve space in .data        **
0003                            ****************************************
0004 000000                             .data
0005 000000 00000000                    .space  0CCh
0006
0007                            ****************************************
0008                            **        Assemble into .text         **
0009                            ****************************************
0010 000000                             .text
0011 000000 00800000                    ABSI    R0
0012
0013                            ****************************************
0014                            **        Assemble into .data         **
0015                            ****************************************
0016 0000CC                     table:  .data
0017 0000CC FFFFFFFF                    .word   -1     ; Assemble 32-bit
0018                                                   ; constant into .data
0019 0000CD 000000FF                    .byte   0FFh   ; Assemble 8-bit
0020                                                   ; constant into .data
0021 0000CE 08010000                    LDI     R0,R1  ; Assemble code into
0022                                                   ; .data
0023
0024                            ****************************************
0025                            **        Assemble into .text         **
0026                            ****************************************
0027 000001                             .text
0028 000001 08010000                    LDI     R0,R1
0029
0030                            ****************************************
0031                            **    Resume assembling into .data    **
0032                            **    at address 0CFh                 **
0033                            ****************************************
0034 0000CF                             .data
```

Syntax **.end**

Description The .end directive is an optional directive that terminates assembly. It should be the last source statement of a program. The assembler will ignore any source statements that follow an .end directive.

Note that this directive has the same effect as an end-of-file.

> **Caution:**
>
> **Do not use the .end directive to terminate a macro; use the $ENDM macro directive instead.**

Example This example shows how the .end directive terminates assembly. If any source statements followed the .end directive, the assembler would ignore them.

```
0001 000000               Text_Start:    .text
0002 000000 0000000A                     .byte     0Ah
0003 000001 0000AAAA                     .word     0AAAAh
0004 000002 41414141                     .string   "AAAAAAAA"
     000003 41414141
0005                                     .end
```

Syntax **.even**

Description The .even directive aligns the section program counter on the next full word. When you use the .field directive, you can follow it with the .even directive. This forces the assembler to write out a partially filled word before initializing fields in the next word. The assembler will fill the unused bits with 0s. If the SPC is already on a word boundary (no word is partially filled), then .even has no effect.

Example Here's an example of the .even directive. Word 0 is initialized with several fields; the .even directive causes the next field to be placed in word 1.

```
0001                         ********************************
0002                         *   Initialize a 2-bit field   *
0003                         ********************************
0004 000000 00000003               .field    03h,2
0005
0006                         ********************************
0007                         *   Initialize a 5-bit field   *
0008                         ********************************
0009 000000 0000002F               .field    0Bh,5
0010
0011                         ********************************
0012                         *      Write out the word      *
0013                         ********************************
0014 000001                        .even
0015
0016                         ********************************
0017                         *     This field is in the     *
0018                         *           next word          *
0019                         ********************************
0020 000001 00000007               .field    07h,3
```

Figure 5-10 shows how this example initializes word 0. The first 7 bits are initialized by .field directives; the remaining bits are set to 0 by the .even directive.

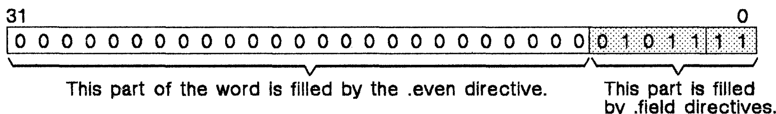


Figure 5-10. An Example of the .even Directive

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

Description The .field directive initializes 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, *size* must be 32.
- The *size* is an optional parameter; it specifies a number from 1–32, which is the number of bits the field consists of. If you do not specify a size, the assembler uses a default size of 32 bits. Note that the assembler will truncate the value if you specify a field that is not wide enough to contain the value. For example, .field 3,1 will cause the assembler to truncate the value 3 to 1; the assembler will also print an error message.

Successive field directives pack values into the specified number of bits in the current word. Fields are packed starting at the least significant part of the word, moving towards the most significant part as more fields are added. If the assembler encounters a field size that will not fit in the current word, it writes out the word, increments the SPC, and begins packing fields into the next word.

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

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

Example This example shows how fields are packed into a word. Notice that the SPC does not change until a word is filled and the next word is begun.

```
0001                          ********************************
0002                          *   Initialize a 24-bit field   *
0003                          ********************************
0004 000000 00BBCCDD                 .field    0BBCCDDh,24
0005
0006                          ********************************
0007                          *   Initialize a 5-bit field    *
0008                          ********************************
0009 000000 0ABBCCDD                 .field    0Ah,5
0010
0011                          ********************************
0012                          *   Initialize a 4-bit field    *
0013                          *          (new word)           *
0014                          ********************************
0015 000001 0000000C                 .field    0Ch,4
0016
0017                          ********************************
0018                          *   Initialize a 3-bit field    *
0019                          ********************************
0020 000001 0000001C          x:     .field    01h,3
0021
0022                          ********************************
0023                          *   Initialize a 32-bit relo-   *
0024                          *   catable field in the next   *
0025                          *            word               *
0026                          ********************************
0027 000002 00000001'               .field    x
```

Figure 5-11 (page 5-27) shows how the directives in this example affect memory.

Word 31 0 Code

(a) 0 | 0 0 1 1 1 0 1 1 1 1 0 0 1 1 0 0 1 1 0 1 1 1 0 1 | .field 0BBCCDDh, 24

24-bit field

(b) 0 | 0 1 0 1 0 | 1 0 1 1 1 0 1 1 1 1 0 0 1 1 0 0 1 1 0 1 1 1 0 1 | .field 0Ah, 5

5-bit field

(c) 0 | 0 0 0 | 0 1 0 1 0 | 1 0 1 1 1 0 1 1 1 1 0 0 1 1 0 0 1 1 0 1 1 1 0 1 | .field 0Ch, 4

1 | 1 1 0 0 |

4-bit field

(d) 1 | 0 0 1 | 1 1 0 0 | .field 01h, 3

3-bit field

(e) 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 1 | 1 1 0 0 | .field x

2 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 |

Figure 5-11. An Example of the .field Directive

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

Description The .float directive places the floating-point representation of one or more floating-point constants into successive words in the current section. Each *value* must be a floating-point constant, or a symbol that has been equated to a floating-point constant. Each constant is converted to a floating-point value in TMS320C30 single-precision (32-bit) format.

Example Here are some examples of the .float directive.

```
0001 000000 53FBA6AF       .float  -1.0e25
0002 000001 01400000       .float  3
0003 000002 06760000       .float  123, 0.5
     000003 FF000000
0004        01490FCF PI:   .set    3.14159
0005        012dF94C E:    .set    2.71828
0006 000004 01490FCF       .float  PI,E
            012dF94C
```

Syntax

.global *symbol₁ [,..., symbolₙ]*

.def *symbol₁ [,..., symbolₙ]*

.ref *symbol₁ [,..., symbolₙ]*

Description

The .global, .def, and .ref directives identify symbols that can be referenced externally.

- The **.def** directive identifies a symbol that is defined in the current module and can be accessed by other files. The assembler will place 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 the symbol's definition.
- The **.global** directive acts as a .ref or a .def, as needed.

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

A symbol may be declared global for two reasons:

1) If the symbol is *not defined in the current source module* (this includes macro, .copy, and include files), then 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.

2) If the symbol *is defined in the current module,* then the .global or .def directive declares that the symbol and its definition can be used externally in other modules. These types of references are resolved at link time.

Example

This example uses 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:

```
0001                              ; Global symbol defined in this file
0002                                      .global    Init
0003                              ; Global symbols defined in file2.lst
0004                                      .global    x,y,z
0005 000000                      Init:                      ; Symbol definition
0006 000000 08010000                      LDI        R0,R1
0007 000001 00000000!                     .word      x
0008                              ;         .
0009                              ;         .
0010                              ;         .
0011                                      .end
```

file2.lst:

```
0001                              ; Global symbols defined in this file
0002                                      .global    x,y,z
0003                              ; Global symbol defined in file1.lst
0004                                      .global    Init
0005                              ; Symbol definitions
0006        00000001              x:      .set       1
0007        00000002              y:      .set       2
0008        00000003              z:      .set       x + y
0009 000000 00000000!                     .word      Init
0010                              ;         .
0011                              ;         .
0012                              ;         .
0013                                      .end
```

file3.lst:

```
0001                              ; Global symbol defined in this file
0002                                      .def       Init
0003                              ; Global symbols defined in file4.lst
0004                                      .ref       x,y,z
0005 000000                      Init:                      ; Symbol definition
0006 000000 08010000                      LDI        R0,R1
0007 000001 00000000!                     .word      x
0008                              ;         .
0009                              ;         .
0010                              ;         .
0011                                      .end
```

file4.lst:

```
0001                              ; Global symbols defined in this file
0002                                      .def       x,y,z
0003                              ; Global symbol defined in file3.lst
0004                                      .ref       Init
0005                              ; Symbol definitions
0006        00000001              x:      .set       1
0007        00000002              y:      .set       2
0008        00000003              z:      .set       x + y
0009 000000 00000000!                     .word      Init
0010                              ;         .
0011                              ;         .
0012                              ;         .
0013                                      .end
```

Syntax **.hword** *value*$_1$ *[,..., value$_n$]*

Description The .hword directive places one or more 16-bit values into consecutive words in the current section. Each *value* may be either:

- An expression which the assembler evaluates and treats as a 16-bit signed number.
- A character string enclosed in double quotes. Each character represents a separate value.

Values are not packed or sign extended; each value occupies the least significant 16 bits of a full 32-bit word.

The assembler truncates any value that is greater than 16 bits. The .hword directive can have up to 100 operands, but they must fit on a single line.

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

Example This example assembles several 16-bit values into words in the current section. The label `vlist` has the value 6Ah, which is the location of the first initialized word.

```
0003 00006A 0000000A  vlist:  .hword  10,-1,"abc",'ab'
     00006B 0000FFFF
     00006C 00000061
     00006D 00000062
     00006E 00000063
     00006F 00006261
```

Syntax

.if *well-defined expression*
code to assemble if expression is true
.else
code to assemble if expression is false
.endif

Description Three directives provide conditional assembly:

- The **.if** directive marks the beginning of a conditional block. The *expression* is a required parameter.
  - If this expression evaluates to *true* (a nonzero value), then the assembler assembles the code that follows it (up to an .else or .endif).
  - If this expression evaluates to *false* (0), then the assembler assembles code that follows an .else (if present) or an .endif (if no .else is present).
- The **.else** directive identifies a block of code that is assembled when the if-expression evaluates to false (0). This directive is optional in the conditional block; if an expression is false and there is no .else statement, then the assembler continues with the code that follows the .endif.
- The **.endif** directive terminates a conditional block.

Example Here are some examples of conditional assembly:

```
0001        00000001  sym1    .set    1
0002        00000002  sym2    .set    2
0003        00000003  sym3    .set    3
0004        00000004  sym4    .set    4
0005                  If_4:   .if     sym4 = sym2 * sym2
0006 000003 00000004          .byte   sym4              ; Equal values
0007                          .else
0008                          .byte   sym2 * sym2       ; Unequal values
0009                          .endif
0010                  If_5:   .if     sym1 <= 10
0011 000004 0000000A          .byte   10                ; Less than/equal
0012                          .else
0013                          .byte   sym1              ; Greater than
0014                          .endif
0015                  If_6:   .if     sym3 * sym2 != sym4 + sym2
0016                          .byte   sym3 * sym2       ; Unequal values
0017                          .else
0018 000005 00000006          .byte   sym4 + sym2       ; Equal values
0019                          .endif
```

Syntax

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

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

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

Description

The .int, .long, and .word directives are equivalent. These directives place one or more values into consecutive 32-bit fields in the current section. Each *value* is either:

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

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.

Example 1

This example uses the .int directive to initialize words. Notice that the symbol `symptr` puts the symbol's address in the object code and generates a relocatable reference (indicated by the ' character appended to the object word).

```
0005 000070 08010000    symptr  LDI    R0,R1
0006 000071 0000000A            .int   10,symptr,-1,"abc",'abc'
     000072 00000070'
     000073 FFFFFFFF
     000074 00000061
     000075 00000062
     000076 00000063
     000077 00636261
```

Example 2

This example initializes two 32-bit fields and defines `DAT1` to point to the first location. The contents of the resulting 32-bit fields are 0FFFFABCDh and 141h.

```
0001 000000 FFFFABCD    DAT1:   .long    0FFFFABCDH,'A'+100h
     000001 00000141
```

Example 3

This example initializes five words. The symbol `WordX` points to the first word.

```
0001 000000 00000C80   WordX: .word 3200,1+'AB',-'AF',0F410h,'A'
     000001 00004242
     000002 FFFFB9BF
     000003 0000F410
     000004 00000041
```

Syntax

.length *page length*

.width *page width*

Description

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

- Default length: 60 lines
- Minimum length: 20 lines
- Maximum length: 32,767 lines

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

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

Note that 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

This example sets the page length and the page width to various values.

```
TMS320C30 Assembler     Version 1.0, 87.089          Thu May 28 14:44:06 1987
(c) Copyright 1987, Texas Instruments Inc.
*****  Length and Width  *****                                    PAGE    1

0002
0003                 *************************************************
0004                 *************************************************
0005                 **      The page length is limited to 60      **
0006                 **      lines per page. The page width is     **
0007                 **      limited to 80 characters per line.    **
0008                 *************************************************
0009                 *************************************************
0010
0011 000000                 .length  60
0012 000000                 .width   80
0013
0014                 *************************************************
0015                 *************************************************
0016                 **      The page length is limited to 50      **
0017                 **      lines per page. The page width is     **
0018                 **      limited to 250 characters per line.   **
0019                 *************************************************
0020                 *************************************************
0021
0022 000000                 .length  50
0023 000000                 .width   250
```

Syntax

.list

.nolist

Description

The .nolist directive suppresses the source listing output until a .list directive is encountered. The .list directive tells the assembler to resume printing the source listing after it has been stopped by a .nolist directive. By default, the assembler behaves as if a .list directive has been specified. The .nolist directive can be used to reduce assembly time and the size of the source listing; it is frequently used in macro definitions to inhibit 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. At the beginning of an assembly, the assembler acts as if it has assembled a .list directive.

Note:

If you don't request a listing file when you invoke the assembler, the assembler ignores the .list directive.

Example

This example uses the .copy directive to insert source statements from another file. The first time the .copy directive is encountered, the assembler lists the copied source lines in the listing file. The second time .copy is encountered, the assembler does not list the copied source lines because a .nolist directive was assembled. Note that the .nolist, the second .copy, and .list directives do not appear in the listing file; note also that the line counter is incremented even when source statements are not listed.

Source file:

```
        .copy   byte.asm
        .nolist
        .copy   byte.asm
        .list
* Back in original file
        .string "Done"
```

Listing file:

```
 0001                              .copy   byte.asm
A0001                        * In byte.asm (copy file)
A0002 000000 00000020              .byte    32, 1+'A'
      000001 00000042
 0006                        * Back in original file
 0007 000004 656E6F44              .string "Done"
```

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

Description The .mlib directive provides the assembler with the name of a macro library. A macro library is a collection of files that contain macro definitions. These files are bound into a single file (called an archive or library) by the archiver. Each file in a macro library may contain one macro definition that corresponds to the name of the file.

Note that:

- Macro library members must be **source** files (not object files).
- The filename of a macro library member must be the same as the macro name and its extension must be .asm.

The *filename* must follow host operating system conventions; it may be enclosed in double quotes. You can specify a full pathname (for example, `.mlib C:\dsp\macs.lib`). If you do not specify a full pathname, the assembler searches for the file in:

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

For more information about the -i option and the environment variable, see Section 4.3, Specifying Alternate Directories for Assembler Input, on page 4-4.

When the assembler encounters an .mlib directive, it opens the library and creates a table of its 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 manner as other macros, but it does not place the source code into the listing. Only macros that are actually called from the library are extracted.

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

incl.asm

```
* Macro for incrementing
incl    $MACRO    REG
        ADDI      1,:REG:
        $ENDM
```

decl.asm

```
* Macro for decrementing
decl    $MACRO    REG
        SUBI      1,:REG:
        $ENDM
```

Use the archiver to create a macro library:

```
ar30 -a mac incl.asm decl.asm
```

Now you can use the .mlib directive to reference the macro library and call the `incl` and `decl` macros:

```
.mlib  "mac.lib"
incl   R0          ; Macro call
decl   R1          ; Macro call
```

Syntax **.mlist**

.mnolist

Two directives provide you with the ability to control the listing of macro expansions in the listing file:

- The **.mlist** directive allows macro expansions in the listing file.
- The **.mnolist** directive inhibits macro expansions in the listing file.

By default, all macro expansions are listed. As the example below shows, the line counters for macro expansion lines are preceded with an exclamation mark (!). The line counter restarts counting at 1 during a macro expansion; it resumes counting from its previous value when the macro expansion is complete.

Example This example defines a macro named str_3. The first time the macro is called, the macro expansion is listed (by default). The second time the macro is called, the macro expansion is not listed because a .mnolist directive was assembled. The third time the macro is called, the macro expansion is again listed because a .mlist directive was assembled.

```
 0001                          str_3    $MACRO       parm1,parm2,parm3
 0002                                   .string     :parm1:,:parm2:,:parm3:
 0003                                   $END
 0004
 0005                                   str_3       "red","green","blue"
!0001 000000 67646572                   .string     "red","green","blue"
      000001 6E656572
      000002 65756C62
 0006                                   .mnolist
 0007                                   str_3       "Socrates","Plato","Aristotle"
 0008                                   .mlist
 0009                                   str_3       "Huron","Superior","Michigan"
!0001 000009 6F727548                   .string     "Huron","Superior","Michigan"
      00000A 7075536E
      00000B 6F697265
      00000C 63694D72
      00000D 61676968
      00000E 0000006E
```

Syntax **.option** *option list*

Description The .option directive selects several options for the assembler output listing. The *option list* is a list of options separated by commas; each option selects a listing feature. Valid options include:

B Limit the listing of .byte directives to one line.
F Reset the B, H, L, M, and T options.
H Limit the listing of .hword directives to one line.
L Limit the listing of .long, .int, and word directives to one line.
M Limit the listing for a macro expansion to a single line.
T Limit the listing of .string directives to one line.
X Produce a symbol cross-reference listing.

Options **are not** case sensitive.

Example This example limits the listings of the .byte, .hword, .long, .word, .int, and .string directives to one line each.

```
0001                          ***************************************
0002                          *  Limit the listing of .byte, .hword, *
0003                          *  .int, .word, .long, and .string     *
0004                          *  directives to one line each         *
0005                          ***************************************
0006                                   .option   B,H,L,T
0007 000000 000000BD                   .byte     -'C',0B0h,5
0008 000003 0000002E                   .hword    56q,0AAAAh
0009 000005 AABBCCDD                   .long     0AABBCCDDh,536+'A'
0010 000007 000015AA                   .word     5546,78h
0011 000009 00000015                   .int      010101b,356q,85
0012 00000C 65747845                   .string   "Extended Registers"
0013
0005                          ***************************************
0015                          *     Reset the listing options       *
0005                          ***************************************
0017 000011                            .option   F
0018 000011 000000BD                   .byte     -'C',0B0h,5
     000012 000000B0
     000013 00000005
0019 000014 0000002E                   .hword    56q,0AAAAh
     000015 0000AAAA
0020 000016 AABBCCDD                   .long     0AABBCCDDh,536+'A'
     000017 00000259
0021 000018 000015AA                   .word     5546,78h
     000019 00000078
0022 00001A 00000015                   .int      010101b,356q,85
     00001B 000000EE
     00001C 00000055
0023 00001D 65747845                   .string   "Extended Registers"
     00001E 6465646E
     00001F 67655220
     000020 65747369
     000021 00007372
```

Syntax .page

Description The .page directive produces a page eject in the listing file. The source statement is not printed in the source listing, but the line counter is incremented. Using the .page directive to divide a source listing into logical divisions improves program readability.

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

Source file:

```
.title    "****  An example of the .page directive   ****"
.string   "Page 1"
.page          ; The directive won't be printed
.string   "Page 2"
```

Listing file:

```
TMS320C30 Assembler     Version 1.00, 87.089           Thu May 28 14:51:38 1987
(c) Copyright 1987, Texas Instruments Inc.
****  An example of the .page directive  ****                            PAGE 1

0002 000000 65676150          .string    "Page 1"
     000001 00003120
TMS320C30 Assembler     Version 1.00, 87.089           Thu May 28 14:51:38 1987
(c) Copyright 1987, Texas Instruments Inc.
****  An example of the .page directive  ****                            PAGE 2

0004 000002 65676150          .string    "Page 2"
     000003 00003220
```

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

Description The .sect directive defines named sections that are used like the default .text and .data sections. The .sect directive begins assembling source code into the named section. Named sections can be used for data or code that must be allocated into memory separately from .text or .data.

The *section name* identifies a section that the assembler assembles code into. The *name* is significant to 8 characters and must be enclosed in double quotes.

Note that the .asect directive is similar to the .sect directive; however, .asect creates a named section that has *absolute addresses*. If you use the .asect directive to define an absolute named section, you can use the .sect directive later in the program to continue assembling code into the absolute section.

Section 3 provides additional information about named sections.

Example This example defines a section, `Sym_Defs`, and assembles code into it.

```
0001                           ***************************************************
0002                           **     Begin assembling into .text section       **
0003                           ***************************************************
0004 000000                            .text
0005 000000 07020001                   LDF      R1,R2    ; Assembled into .text
0006 000001 07040003                   LDF      R3,R4    ; Assembled into .text
0007
0008                           ***************************************************
0009                           **    Begin assembling into Sym_Defs section     **
0010                           ***************************************************
0011 000000                            .sect    "Sym_Defs"
0012 000000 0148F5C2                   .float   3.14
0013 000001 0000000F                   .hword   0Fh
0014 000002 07060005                   LDF      R5,R6    ; Assembled into Sym_Defs
0015
0016                           ***************************************************
0017                           **     Resume assembling into .text section      **
0018                           ***************************************************
0019 000000                            .text
0020 000002 080A0009                   LDI      AR1,AR2 ; Assembled into .text
0021 000003 00000003                   .byte    3,4
     000004 00000004
0022
0023                           ***************************************************
0024                           **    Resume assembling into Sym_Defs section    **
0025                           ***************************************************
0026 000003                            .sect    "Sym_Defs"
0027 000003 AABBCCDD                   .long    0aabbccddh
0028
```

Syntax

symbol **.set** *value*

Description

The .set directive assigns a value to a symbol. The symbol can then be used in place of the value in source statements. This allows you to equate meaningful names with constants, registers, and other values.

- The *symbol* 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.

```
0001                      *************************************
0002                      **   Equate symbol FP to register   **
0003                      **   AR3 and use it instead of the  **
0004                      **   register                       **
0005                      *************************************
0006          0000000B    FP       .set      AR3
0007 000000   0840C300             LDI       *FP,R0
0008
0009                      *************************************
0010                      **  Set symbol count to an integer  **
0011                      **  expression and use it as an     **
0012                      **  immediate operand               **
0013                      *************************************
0014          00000035    count    .set      100/2 + 3
0015 000001   08600035             LDI       count,R0
0016
0017                      *************************************
0018                      **   Set symbol symtab to a relo-   **
0019                      **   catable expression and use it  **
0020                      **   as a relocatable operand       **
0021                      *************************************
0022 000002   0000000A    label    .word     10
0023          00000003'   symtab   .set      label+1
0024 000003   08200003+            LDI       @symtab,R0
0025
0026                      *************************************
0027                      **   Set symbol PI to a floating-   **
0028                      **   point constant and use it as   **
0029                      **   an operand                     **
0030                      *************************************
0031          01490FCF    PI       .set      3.14159
0032 000004   01490FCF             .float    PI
0033
0034                      *************************************
0035                      **   Set symbol nsyms equal to the  **
0036                      **   symbol count and use it as you **
0037                      **   would use count                **
0038                      *************************************
0039          00000035    nsyms     .set     count
0040 000005   08600035              LDI      nsyms,R0
```

Syntax **.space** *size in words*

Description The .space directive reserves *size* number of words in the current section and fills them with 0s. The section program counter is incremented to point to the word following the reserved space.

The .space directive is equivalent to *size* number of `.word 0` directives.

Example This example reserves 100 0-filled words in the .text section. Note that the SPC equals 03h before the .space directive is assembled; after the .space directive is assembled, the SPC is incremented to equal 067h.

```
0001                        ********************************************
0002                        *       Begin assembling into .text        *
0003                        ********************************************
0004 000000                             .text
0005 000000 0000000A                    .word    0Ah, 0Bh
     000001 0000000B
0006 000002 00004230                    .string  "AR0"
0007                        ********************************************
0008                        *  Reserve a block of 100 words in .text   *
0009                        ********************************************
0010 000003 00000000        Sp_X:       .space   100
0011 000067 0000000C                    .word    0Ch    ; Still in .text
0012 000068 00000003'                   .word    Sp_X
```

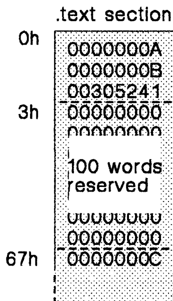


Figure 5-12. An Example of the .space Directive

Syntax **.string** *"string₁" [,..., "stringₙ"]*

Description The .string directive places 8-bit characters from a character string into the current section. The data is packed so that each word contains four 8-bit values. Each *string* is either:

- An expression which the assembler will evaluate and treat as a 32-bit signed number.
- A character string enclosed in double quotes. Each character represents a separate value.

Values are packed into words starting with the least significant byte of the word and moving toward the most significant portion as more bytes are added. Any unused space is padded with null bytes (0s).

The assembler truncates any values that are greater than 8 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 first word that is initialized.

Example This example places 8-bit values into words in the current section.

```
0001 000000 44434241   Str_3:  .string   "ABCD"
0002 000001 54535251           .string   51h, 52h, 53h, 54h
0003 000002 73756F48           .string   "Houston"
     000003 006E6F74
0004 000004 00000030           .string   36 + 12
```

Syntax **.text**

Description

The .text directive tells the assembler to begin assembling into the .text section, which normally 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.

Note that 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 specify one of the other initialized-section directives (.data, .sect, or .asect).

For more information about COFF sections, see Section 3.

Example

This example assembles code into the .text and .data sections. The .text section contains bytes 1, 2, 3, and 4, and the .data section contains bytes 5, 6, 7, and 8.

```
0001                          ****************************************
0002                          ** Begin assembling into .data section **
0003                          ****************************************
0004 000000                            .data
0005 000000 00000005                   .byte   5,6
     000001 00000006
0006
0007                          ****************************************
0008                          ** Begin assembling into .text section **
0009                          ****************************************
0010 000000                            .text
0011 000000 00000001                   .byte   1
0012 000001 00000002                   .byte   2,3
     000002 00000003
0013
0014                          ****************************************
0015                          **     Resume assembling into .data    **
0016                          ****************************************
0017 000002                            .data
0018 000002 00000007                   .byte   7,8
     000003 00000008
0019
0020                          ****************************************
0021                          **    Resume assembling into .text     **
0022                          ****************************************
0023 000003                            .text
0024 000003 00000004                   .byte   4
```

Syntax **.title** *"string"*

Description The .title directive supplies a title that is printed in the heading on each listing page. The source statement itself is not printed, but the line counter is incremented. The *string* is a quote-enclosed title of up to 50 characters. If you supply more than 50 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 of a listing, then the first source statement must contain a .title directive.

Example This example prints the title `*** Floating Point Routines ***` in the page headings of the source listing.

Source statement:

```
.title    "*** Floating Point Routines ***"
```

Listing file:

```
 TMS320C30 Assembler     Version 1.00, 87.089          Tue Apr 21 11:39:03 198
 (c) Copyright 1987, Texas Instruments Inc.
***  Floating Point Routines   ***                                 PAGE     1
                                          .
                                          .
                                          .
                                          .
                                          .
                                          .
 TMS320C30 Assembler     Version 1.00, 87.089          Tue Apr 21 11:39:03 198
 (c) Copyright 1987, Texas Instruments Inc.
***  Floating Point Routines   ***                                 PAGE     2
                                          .
                                          .
                                          .
                                          .
                                          .
                                          .
```

Syntax *symbol* **.usect** *"section name", size in words*

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 that you're reserving space for.
- The *section name* must be enclosed in double quotes; only the first 8 characters are significant. This parameter names the uninitialized section.
- The *size* is an expression that defines the number of words that will be reserved in section *name*.

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

You can repeat the .usect directive to define more than one variable in the specified section. Variables which can be located contiguously in memory can be defined in the same section by using multiple .usect directives with the same section name.

For more information about COFF sections, see Section 3.

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

Figure 5-13 shows how this example reserves space in two uninitialized sections, `var1` and `var2`.

```
0001                            ***********************************
0002                            *      Assemble into .text        *
0003                            ***********************************
0004 000000                             .text
0005 000000 08010000                    LDI      R0,R1
0006
0007                            ************************************
0008                            *     Reserve 1 word in var1       *
0009                            ************************************
0010 000000                     ptr      .usect    "var1", 1
0011
0012                            ************************************
0013                            *  Reserve 100 more words in var1  *
0014                            ************************************
0015 000001                     array    .usect   "var1", 100
0016
0017 000001 08020001                    LDI      R1,R2         ; Still in .text
0018
0019                            ************************************
0020                            *   Reserve 50 more words in var1  *
0021                            ************************************
0022 000002                     dflag   .usect    "var1", 50
0023
0024 000002 08030002                   LDI       R2,R3         ; Still in .text
0025
0026                            ************************************
0027                            *    Reserve 100 words in var2     *
0028                            ************************************
0029 000000                     vec     .usect    "var2", 100
0030
0031 000003 08200000                   LDI       @vec,R0       ; Still in .text
0032
0033                            ************************************
0034                            * Declare an external .usect symbol *
0035                            ************************************
0036                                   .global   array
```

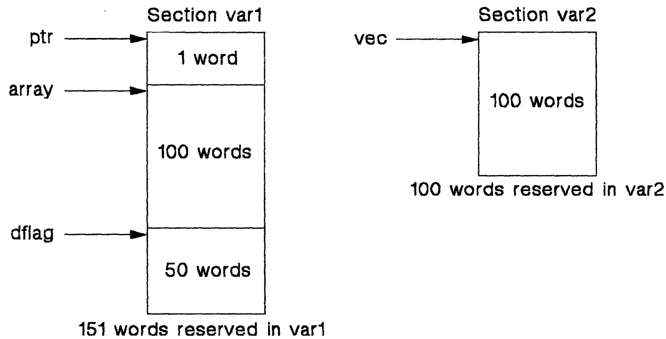


Figure 5-13. An Example of the .usect Directive

Section 6

Instruction Set

The TMS320C30 supports a base set of general-purpose instructions as well as arithmetic-intensive instructions that are particularly suited for digital signal processing and other numeric-intensive applications.

This section does not cover topics such as opcodes or instruction timing; the *Third-Generation TMS320 User's Guide* discusses the instruction set in detail. The *Third-Generation TMS320 User's Guide* also contains an alphabetical presentation which is similar to the directives reference that begins on page 5-13.

This section provides a general summary of the TMS320C30 instruction set:

- Section 6.1 lists the syntax, operation, and description of each instruction.
- Section 6.2 and Section 6.3 summarize and describe optional syntaxes of three-operand instructions and parallel instructions, respectively.
- Section 6.4 through Section 6.8 describe the functional categories of the instruction set.

Section **Page**

6.1 Summary 6-2
6.2 Three-Operand Instructions 6-17
6.3 Parallel Instructions 6-18
6.4 Load and Store Instructions 6-21
6.5 Arithmetic Instructions 6-22
6.6 Logical Instructions 6-22
6.7 Program-Control Instructions 6-23
6.8 Interlocked-Operation Instructions 6-23
6.9 The LDP Instruction 6-24

6.1 Summary

Section 6.1.4 lists the TMS320C30 instruction set alphabetically. Each table entry shows the instruction's syntax and operation, contains a brief description, and shows any optional syntaxes. The key for Section 6.1.4 lists the valid addressing modes that can be used for various operands. Section 6.1.1 summarizes these addressing modes, Section 6.1.2 summarizes the optional syntaxes, and Section 6.1.3 summarizes the condition codes used with conditional instructions. Section 6.1.4 begins on page 6-5.

6.1.1 Addressing Modes

The *Third-Generation TMS320 User's Guide* discusses addressing modes in detail. This section summarizes the addressing modes mentioned in Section 6.1.4.

- ***General addressing modes:***

  Register mode: The operand is a CPU register. For floating-point operations, use an extended register (R0–R7). For integer operations, use any register.

  Short immediate mode: The operand is a 16-bit immediate value. Short immediate operands may be signed integers, unsigned integers, or floating-point values, depending on the instruction.

  Direct mode: The operand is the contents of a 24-bit address, specified by @addr. The 8 MSBs of the address are specified by the DP register; the 16 LSBs are specified by the instruction word. (You can use the LDP instruction to load the page number into the data page pointer register.)

  Indirect mode: An auxiliary register indicates the address of the operand. Table 6-1 lists the various forms that indirect operands may take. The displacement may be specified as a value from 0–255 or as one of the index registers (IR0 or IR1).

  It is not necessary to specify the displacement if it is 1, because the assembler assumes a default displacement of 1. For example, *++ARn is equivalent to *++ARn(1).

- ***Three-operand addressing modes:***

  Register mode: Same as for general addressing modes.

  Indirect mode: Same as for general addressing modes, except the displacement is limited to 0, 1, IR0, or IR1.

- ***Parallel addressing mode:***

  Register mode: The operand is an extended register (R0–R7). In some cases, only R0/R1 or R2/R3 can be used as an operand.

  Indirect mode: Same as for general addressing modes, except the displacement is limited to 0, 1, IR0, or IR1.

- ***Long-immediate addressing mode:***

  The operand is a 24-bit immediate value (usually specified by a label).

- ***Conditional branch addressing mode:***

  Register mode: Same as for general addressing modes; the contents of the register are loaded into the PC.

  PC-relative mode: A signed 16-bit displacement is added to the PC. The destination address is usually specified as a label; the assembler calculates the displacement.

Table 6-1. Indirect Addressing Mode

Operand	Description
***ARn**	Indirect with no displacement
***+ARn(disp)**	Indirect with predisplacement or preindex add
***-ARn(disp)**	Indirect with predisplacement or preindex subtract
***++ARn(disp)**	Indirect with predisplacement or preindex add and modification
***--ARn(disp)**	Indirect with predisplacement or preindex subtract and modification
***ARn++(disp)[%]** †	Indirect with postdisplacement or postindex add and modification
***ARn--(disp)[%]** †	Indirect with postdisplacement or postindex subtract and modification
***ARn++(IR0)B**	Indirect with postindex (IR0) and bit-reversed modification

† Optional circular modification (specified by %)

6.1.2 Optional Syntaxes

The assembler allows a relaxed syntax form for several instructions. These optional forms simplify the assembly language so that you can ignore special-case syntax for some instructions.

- If the source and destination register are the same, you need only specify the register once. Instructions that can use this optional syntax include:

ABSF	FIX	NEGB	NEGI	NOT
ABSI	FLOAT	NEGF	NORM	RND

  For example,
  `ABSI  R0,R0`   *can be written as*   `ABSI R0`

- You can omit the displacement for indirect operands; the assembler will assume a displacement of 1. Instructions that use general addressing modes, three-operand addressing modes, or parallel addressing modes may have indirect address operands. For example,

  `LDI  *AR0++(1),R0`   *can be written as*   `LDI  *AR0++,R0`

- Long-immediate mode operands can be written with an @ symbol. The branch and call instructions can use this optional syntax. For example,

  `BR   label`   *can be written as*   `BR   @label`

6.1.3 Condition Codes

The TMS320C30 supports conditional loads, branches, traps, calls, and returns. These instructions use the condition codes in Table 6-2.

Table 6-2. Condition Codes

Unconditional Compares			
Cond.	**Code**	**Description**	**Flags**
U	00000	Unconditional	don't care
Unsigned Compares			
Cond.	**Code**	**Description**	**Flags**
LO	00001	Lower than	C
LS	00010	Lower or same	C OR Z
HI	00011	Higher than	$\overline{C}$ AND $\overline{Z}$
HS	00100	Higher or same	$\overline{C}$
EQ	00101	Equal	Z
NE	00101	Not equal	$\overline{Z}$
Signed Compares			
Cond.	**Code**	**Description**	**Flags**
LT	00111	Less than	N
LE	01000	Less than or equal	N OR Z
GT	01001	Greater than	$\overline{N}$ AND $\overline{Z}$
GE	01010	Greater than or equal	$\overline{N}$
EQ	00101	Equal	Z
NE	00101	Not equal	$\overline{Z}$
Compare to Zero			
Cond.	**Code**	**Description**	**Flags**
Z	00101	Zero	Z
NZ	00110	Not zero	$\overline{Z}$
P	01001	Positive	$\overline{N}$ AND $\overline{Z}$
N	00111	Negative	N
NN	01011	Nonnegative	$\overline{N}$
Compare to Condition Flags			
Cond.	**Code**	**Description**	**Flags**
NN	01011	Nonnegative	$\overline{N}$
N	00111	Negative	N
NZ	00110	Nonzero	$\overline{Z}$
Z	00101	Zero	Z
NV	01100	No overflow	$\overline{V}$
V	01101	Overflow	V
NUF	01110	No underflow	$\overline{UF}$
UF	01111	Underflow	UF
NC	00100	No carry	$\overline{C}$
C	00001	Carry	C
NLV	10000	No latched overflow	$\overline{LV}$
LV	10001	Latched overflow	LV
NLUF	10010	No latched floating-point underflow	$\overline{LUF}$
LUF	10011	Latched floating-point underflow	LUF
ZUF	10100	Zero or floating-point underflow	Z OR UF

6.1.4 Instruction Set Summary Table

Syntax	Description
ABSF *Src,Rn* [**ABSF** *Rn*]	**Absolute Value of a Floating-Point Number** Operation: |Src| → Rn Load the absolute value of a floating-point number into an extended-precision register.
ABSI *Src,Dreg* [**ABSI** *Dreg*]	**Absolute Value of an Integer** Operation: |Src| → Dreg Load the absolute value of an integer into a register.
ADDC *Src,Dreg*	**Add Integers with Carry** Operation: Src + Dreg + C → Dreg Add the source, the contents of the destination register, and the carry bit together, and store the sum in the destination register. The operands are signed integers.
ADDC3 *Src1,Src2,Dreg* [**ADDC** *Src1,Src2,Dreg*]	**Add Integers with Carry (3-Operand)** Operation: Src1 + Src2 + C → Dreg Add the two source operands and the carry bit together, and store the sum in the destination register. The operands are signed integers.
ADDF *Src,Rn*	**Add Floating-Point Values** Operation: Src + Rn → Rn Add the source operand to the contents of an extended-precision register, and store the sum into the register. The operands are floating-point numbers.
ADDF3 *Src1,Src2,Rn* [**ADDF** *Src1,Src2,Rn*]	**Add Floating-Point Values (3-Operand)** Operation: Src1 + Src2 → Rn Add the two source operands together and store the sum in the destination register. The operands are floating-point numbers.
ADDI *Src,Dreg*	**Add Integers** Operation: Src + Dreg → Dreg Add the source operand to the contents of the destination register and store the sum in the destination register. The operands are signed integers.
ADDI3 *Src1,Src2,Dreg* [**ADDI** *Src1,Src2,Dreg*]	**Add Integers (3-Operand)** Operation: Src1 + Src2 → Dreg Add the two source operands together and store the sum in the destination register. The operands are signed integers.

Note: Optional syntaxes are shown in [*brackets*].

Key:

Src – General addressing modes
Src1 – Three-operand addressing modes
Src2 – Three-operand addressing modes
Csrc – Conditional branch addressing modes
Sreg – Register mode (any register)
Count – Shift value (general addressing modes)
Dreg – Register mode (any register)
Rn – Register mode (R0–R7)
Daddr – Destination memory address
ARn – Auxiliary register n (AR0–AR7)
Addr – 24-bit immediate address (label)
Cond – Condition code (see Table 6-2, pg. 6-4)

Syntax	Description
AND3 *Src1,Src2,Dreg* [**AND** *Src1,Src2,Dreg*]	**Bitwise Logical AND (3-Operand)** Operation: Src1 AND Src2 → Dreg Perform a bitwise logical AND of the two source operands and store the result in the destination register. All the operands are unsigned integers.
ANDN *Src,Dreg*	**Bitwise Logical AND with Complement** Operation: Dreg AND ~Src → Dreg Perform a bitwise logical AND of the destination register and the bitwise logical complement of the source operand, and store the result into the destination register. Both operands are unsigned integers.
ANDN3 *Src1,Src2,Dreg* [**ANDN** *Src1,Src2,Dreg*]	**Bitwise Logical ANDN (3-Operand)** Operation: Src1 AND 7E Src2 → Dreg Perform a bitwise logical AND of source operand 1 and the bitwise logical complement of the source operand 2, and store the result into the destination register. All the operands are unsigned integers.
ASH *Count,Dreg*	**Arithmetic Shift** Operation: If Count ≥ 0 Dreg << Count → Dreg Else Dreg >> |Count| → Dreg If Count is greater than or equal to 0, left shift the contents of the destination register by Count. Low-order bits are filled with 0s, and high-order bits are shifted out through the carry bit. If Count is less than 0, right shift the contents of the destination register by the absolute value of Count. High-order bits are sign ex tended, and low-order bits are shifted out through the carry bit. Both operands are signed integers.
ASH3 *Count,Src,Dreg* [**ASH** *Count,Src,Dreg*]	**Arithmetic Shift (3-Operand)** Operation: If Count ≥ 0 Src << Count → Dreg Else Src >> |Count| → Dreg If Count is greater than or equal to 0, left shift the source operand by Count. Low-order bits will be filled with 0s, and high-order bits are shifted out through the carry bit. If Count is less than 0, right shift the contents of the destination register by the absolute value of Count. High-order bits are sign extended, and low-order bits are shifted out through the carry bit. The shifted value is stored in the destination register. Both operands are signed integers.
B*cond* *Csrc* [**B*cond*** *@Csrc*]	**Branch Conditionally (standard)** Operation: If *cond* = true If Csrc is a register, Csrc → PC If Csrc is an immediate value, Csrc + PC + 1 → PC Else continue Perform a branch if the condition is true. If Csrc is a register or a label, its contents are loaded into the PC. If Csrc is a 16-bit immediate value, it is added to the PC. You can precede labels with an @ symbol.

Syntax	Description

B*cond*D *Csrc*
[**B*cond*D** *@Csrc*]
	Branch Conditionally (delayed)
	Operation: If *cond* = true
	  If Csrc is a register, Csrc → PC
	  If Csrc is an immediate value, Csrc + PC + 3 → PC
	Else
	  continue
	Perform a branch if the condition is true. If Csrc is a register or a label, its contents are loaded into the PC. If Csrc is a 16-bit immediate value, it is added to the PC. You can precede labels with an @ symbol.

BR *Addr* [**BR** *@Addr*]	**Branch Unconditionally (standard)** Operation: Addr → PC Perform an unconditional branch. The source operand is a label or a 24-bit unsigned immediate value. If the D is specified, the branch is delayed. If the operand is a label, you can precede it with an @ symbol.
BRD *Addr* [**BRD** *@Addr*]	**Branch Unconditionally (delayed)** Operation: Addr → PC Perform an unconditional branch. The source operand is a label or a 24-bit unsigned immediate value. If the D is specified, the branch is delayed. If the operand is a label, you can precede it with an @ symbol.
CALL *Addr* [**CALL** *@Addr*]	**Call Subroutine** Operation: Next PC → *++SP Addr → PC Call a subroutine. If the operand is a label, you can precede it with an @ symbol.

CALL*cond* *Csrc*
[**CALL*cond*** *@Csrc*]
	Call Subroutine Conditionally
	Operation: If *cond* = true
	  Next PC → *++SP
	  If Csrc is a register, Csrc → PC
	  If Csrc is an immediate value, Csrc + PC → PC
	Else
	  continue
	Call a subroutine if the condition is true. If Csrc is a register or a label, its contents are loaded into the PC. If Csrc is a 16-bit immediate value, it is added to the PC. You can precede labels with an @ symbol.

CMPF *Src,Rn*	**Compare Floating-Point Values** Operation: Set flags on Rn - Src Compare the source and destination operands by subtracting the source from the destination and setting the appropriate status bits. The result of the subtraction is not stored – this is a nondestructive compare. Both operands are floating-point numbers.

Note: Optional syntaxes are shown in [*brackets*].

Key:

Src – General addressing modes
Src1 – Three-operand addressing modes
Src2 – Three-operand addressing modes
Csrc – Conditional branch addressing modes
Sreg – Register mode (any register)
Count – Shift value (general addressing modes)
Dreg – Register mode (any register)
Rn – Register mode (R0–R7)
Daddr – Destination memory address
ARn – Auxiliary register n (AR0–AR7)
Addr – 24-bit immediate address (label)
Cond – Condition code (see Table 6-2, pg. 6-4)

<table>
<tr><th>Syntax</th><th>Description</th></tr>
<tr><td>CMPF3 Src2,Src1
[CMPF Src2,Src1]</td><td>Compare Floating-Point Values (3-Operand)
Operation: Set flags on Src1 - Src2
Compare the two source operands by subtracting source 2 from source 1 and setting the appropriate status bits. The result of the subtraction is not stored – this is a nondestructive compare. Both operands are floating-point numbers.</td></tr>
<tr><td>CMPI Src,Dreg</td><td>Compare Integers
Operation: Set flags on Dreg - Src
Compare the source and destination operands by subtracting the source from the destination and setting the appropriate status bits. The result of the subtraction is not stored – this is a nondestructive compare. Both operands are integers.</td></tr>
<tr><td>CMPI3 Src2,Src1
[CMPI Src2,Src1]</td><td>Compare Integers (3-Operand)
Operation: Set flags on Src1 - Src2
Compare the two source operands by subtracting source 2 from source 1 and setting the appropriate status bits. The result of the subtraction is not stored – this is a nondestructive compare. Both operands are integers.</td></tr>
<tr><td>DBcond ARn,Csrc
[DBcond ARn,@Csrc]</td><td>Decrement and Branch Conditionally (standard)
Operation: ARn - 1 → ARn
If cond = true and ARn ≥ 0
If Csrc is a register, Csrc → PC
If Csrc is an immediate value, Csrc + PC + 1 → PC
Else
continue
Decrement the specified auxiliary register and branch if the condition is true and the specified auxiliary register is not zero. If Csrc is a register or a label, its contents are loaded into the PC. If Csrc is a 16-bit immediate value, it is added to the PC. You can precede labels with an @ symbol.</td></tr>
<tr><td>DBcondD ARn,Csrc
[DBcond ARn,@Csrc]</td><td>Decrement and Branch Conditionally (delayed)
Operation: ARn - 1 → ARn
If cond = true and ARn ≥ 0
PC + 3 → PC
If Csrc is a register, Csrc → PC
If Csrc is an immediate value, Csrc + PC + 3 → PC
Else
continue
Decrement the specified auxiliary register and branch if the condition is true and the specified auxiliary register is not zero. If Csrc is a register or a label, its contents are loaded into the PC. If Csrc is a 16-bit immediate value, it is added to the PC. You can precede labels with an @ symbol.</td></tr>
<tr><td>FIX Src,Dreg
[FIX Dreg]</td><td>Convert Floating-Point Value to Integer
Operation fix(Src) → Dreg
Convert a floating-point operand to the nearest integer which is less than or equal to its absolute value and load the result into the destination register.</td></tr>
<tr><td>FLOAT Src,Rn
[FLOAT Rn]</td><td>Convert Integer to Floating-Point Value
Operation: float(Src) → Rn
Convert an integer into a floating-point value and load the result into an extended-precision register.</td></tr>
</table>

Count – Shift value (general addressing modes) **Cond** – Condition code (see Table 6-2, pg. 6-4)

Syntax	Description
IACK *Src*	**Interrupt Acknowledge** Operation: Perform a dummy read operation with IACK = 0. At end of dummy read, set IACK = 1. Perform a dummy read operation with IACK = 0. IACK is set to 1 at the end of the dummy read. This instruction can be used to generate an external interrupt acknowledge. If the specified address is off-chip the processor reads the data at that address. Then, the IACK signal and the address can be used to signal an interrupt acknowledge to external devices. The data read by the processor is not used.
IDLE	**Idle Until Interrupt** Operation: 1 → ST(GIE) Next PC → PC Idle until interrupt Load the next PC value into the PC and idle until an interrupt is received. When an interrupt is received, the contents of the PC are pushed onto the system stack.
LDE *Src,Rn*	**Load Floating-Point Exponent** Operation: Src(exponent) → Rn(exponent) Load the exponent portion of a word into the exponent field of an extended-precision register.
LDF *Src,Rn*	**Load Floating-Point Value** Operation: Src → Rn Load a floating point-value into an extended-precision register.
LDF*cond* *Src,Rn*	**Load Floating-Point Value Conditionally** Operation: If *cond* = true Src → Rn Else Rn is not changed If the specified condition is true, a floating-point value is loaded into an extended-precision register. If the condition is false, the value is not loaded.
LDFI *Src,Dreg*	**Load Floating-Point Value, Interlocked** Operation: Signal interlocked operation Src → Rn The source operand is loaded into the destination register and an interlocked operation is signaled over the XF0 and XF1 pins. The operands are floating-point values.
LDI *Src,Dreg*	**Load Integer** Operation: Src → Dreg Load the contents of the source operand into a register. The source operand is a signed integer.

Note: Optional syntaxes are shown in [*brackets*].

Key:

Src – General addressing modes
Src1 – Three-operand addressing modes
Src2 – Three-operand addressing modes
Csrc – Conditional branch addressing modes
Sreg – Register mode (any register)
Dreg – Register mode (any register)
Rn – Register mode (R0–R7)
Daddr – Destination memory address
ARn – Auxiliary register n (AR0–AR7)
Addr – 24-bit immediate address (label)

Syntax	Description
LDI*cond Src,Dreg*	**Load Integer Conditionally** Operation: If *cond* = true Src → Dreg Else Dreg is not changed If the specified condition is true, the contents of the source operand are loaded into a register. The source operand is an integer. If the condition is false, the source operand is not loaded.
LDII *Src,Dreg*	**Load Integer, Interlocked** Operation: Signal interlocked operation Src → Dreg The source operand is loaded into the destination register and an interlocked operation is signaled over the XF0 and XF1 pins. The operands are signed integers.
LDM *Src,Rn*	**Load Floating-Point Mantissa** Operation: Src(mantissa) → Rn(mantissa) Load the mantissa portion of a word into the mantissa field of an extended-precision register.
LSH *Count,Dreg*	**Logical Shift** Operation: If Count ≥ 0 Dreg << Count → Dreg Else Dreg >> |Count| → Dreg If Count is greater than or equal to zero, left shift the contents of the destination register by Count. Low-order bits are filled with 0s and high-order bits are shifted out through the carry bit. If Count is less than zero, right shift the contents of the destination register by the absolute value of Count. High-order bits are filled with 0s and low-order bits are shifted out through the carry bit. The Count operand is a signed integer; Dreg is an unsigned integer.
LSH3 *Count,Src,Dreg* [**LSH** *Count,Src,Dreg*]	**Logical Shift (3-Operand)** Operation: If Count ≥ 0 Src << Count → Dreg Else Src >> |Count| → Dreg If Count is greater than or equal to zero, left shift the source operand by Count. Low-order bits are filled with 0s and high-order bits are shifted out through the carry bit. The result is stored in the destination register. If Count is less than zero, right shift the source operand by the absolute value of Count. High-order bits are filled with 0s and low-order bits are shifted out through the carry bit. The Count operand is a signed integer; Src is an unsigned integer. The result is stored in the destination register.
MPYF *Src,Rn*	**Multiply Floating-Point Values** Operation: Src × Rn → Rn Multiply the source operand by the contents of an extended-precision register and store the result into the register. Both operands are floating-point numbers.

Syntax	Description
MPYF3 *Src1,Src2,Rn* [**MPYF** *Src1,Src2,Rn*]	**Multiply Floating-Point Values (3-Operand)** Operation: Src1 × Src2 → Rn Multiply the two source operands together and store the result into the extended-precision register. All the operands are floating-point numbers.
MPYI *Src,Dreg*	**Multiply Integers** Operation: Src × Dreg → Dreg Multiply the source operand by the contents of the destination register and store the result in the register. Both operands are 24-bit signed integers; the result is the 32 LSBs of the product.
MPYI3 *Src1,Src2,Dreg* [**MPYI** *Src1,Src2,Dreg*]	**Multiply Integers (3-Operand)** Operation: Src1 × Src2 → Dreg Multiply the two source operands and store the result in the register. All the operands are 24-bit signed integers; the result is the 32 LSBs of the product.
NEGB *Src,Dreg* [**NEGB** *Dreg*]	**Negate Integer with Borrow** Operation: 0 - Src - C → Dreg Load the difference between the source operand, 0, and the carry bit into the destination register. The operands are signed integers.
NEGF *Src,Rn* [*NEGF Rn*]	**Negate Floating-Point Value** Operation: 0 - Src → Rn Load the difference between the source operand and 0 into the extended-precision register. The operands are floating-point numbers.
NEGI *Src,Dreg* [**NEGI** *Dreg*]	**Negate Integer** Operation: 0 - Src → Dreg Load the difference between the source operand and 0 into the destination register. The operands are signed integers.
NOP [**NOP** *Src*]	**No Operation** Operation: No ALU or multiplier operations. ARn is modified if Src is specified in indirect mode. Modify the source operand (if specified), or perform no operation. Src must be an indirect operand.
NORM *Src,Rn* [**NORM** *Rn*]	**Normalize Floating-Point Value** Operation: normalize(Src) → Rn Normalize a floating-point number and load the result into an extended-precision register.
NOT *Src,Dreg* [**NOT** *Dreg*]	**Bitwise Logical Complement** Operation: $\overline{\text{Src}}$ → Dreg Load the bitwise logical complement of the source operand into the destination register.

Note: Optional syntaxes are shown in [*brackets*].

Key:

Src – General addressing modes
Src1 – Three-operand addressing modes
Src2 – Three-operand addressing modes
Csrc – Conditional branch addressing modes
Sreg – Register mode (any register)
Count – Shift value (general addressing modes)
Dreg – Register mode (any register)
Rn – Register mode (R0–R7)
Daddr – Destination memory address
ARn – Auxiliary register n (AR0–AR7)
Addr – 24-bit immediate address (label)
Cond – Condition code (see Table 6-2, pg. 6-4)

Syntax	Description
OR *Src,Dreg*	**Bitwise Logical OR** Operation: Dreg OR Src → Dreg Load the bitwise logical OR of the source and the destination into the destination register. The operands are unsigned integers.
OR3 *Src1,Src2,Dreg* [**OR** *Src1,Src2,Dreg*]	**Bitwise Logical OR (3-Operand)** Operation: Src1 OR Src2 → Dreg Load the bitwise logical OR of the two source operands into the destination register. The operands are unsigned integers.
POP *Dreg*	**Pop Integer from Stack** Operation: *SP-- → Dreg Pop the contents of the top of the system stack into the destination register. The value popped from the stack is an integer.
POPF *Rn*	**Pop Floating-Point Value from Stack** Operation: *SP-- → Rn Pop the contents of the top of the system stack into an extended-precision register. The value popped from the stack is a floating-point number.
PUSH *Sreg*	**Push Integer on Stack** Operation: Sreg → *++SP Push the contents of the source register onto the top of the system stack. The value pushed on the stack is an integer.
PUSHF *Rn*	**Push Floating-Point Value on Stack** Operation: Rn → *++SP Push the contents of an extended-precision register onto the top of the system stack. The value pushed on the stack is a floating-point number.
RETI*cond* [**RETI**]	**Return from Interrupt Conditionally or Unconditionally** Operation: If *cond* = true *SP-- → PC 1 → ST(GIE) Else continue Perform a return from an interrupt routine. If the condition is true or if there is no condition, pop the top of the system stack into the PC and set the global interrupt enable bit to 1.
RETS*cond* [**RETS**]	**Return from Subroutine Conditionally or Unconditionally** Operation: If *cond* = true *SP-- → PC Else continue Perform a return from a subroutine. If the condition is true or missing, pop the top of the system stack into the PC.
RND *Src,Rn* [**RND** *Rn*]	**Round Floating-Point Value** Operation: round(Src) → Rn Round the source operand to the nearest single-precision floating-point number and load it into an extended-precision register.

Syntax	Description
ROL *Dreg*	**Rotate Left** Operation: Dreg rotated left 1 bit → Dreg Rotate the contents of the destination register left one bit and store the result back into the destination register. The carry bit is set to the original value of the MSB.
ROLC *Dreg*	**Rotate Left through Carry** Operation: Dreg rotated left 1 bit through carry → Dreg Rotate the contents of the destination register left one bit through the carry bit and store the result back into the destination register. The carry bit is set to the original value of the MSB and the new LSB value is set to the original value of the carry bit.
ROR *Dreg*	**Rotate Right** Operation: Dreg right-rotated 1 bit through carry bit → Dreg Rotate the contents of the destination register right one bit and store the result back into the destination register. The carry bit is set to the original value of the LSB.
RORC *Dreg*	**Rotate Right through Carry** Operation: Dreg rotated right 1 bit through carry → Dreg Rotate the contents of the destination register right one bit through the carry bit and store the result back into the destination register. The carry bit is set to the original value of the LSB and the new MSB value is set to the original value of the carry bit.
RPTB *Val*	**Repeat Block of Instructions** Operation: Val → RE 1 → ST(RM) next PC → RS Repeat a block of instructions by the number in the RC (repeat count) register. Val is a 24-bit immediate value that is loaded into the repeat end address (RE) register. The RM (repeat mode) status bit is set to 1, and the address of the next instruction is loaded into the repeat start address (RS) register.
RPTS *Val*	**Repeat Single Instruction** Operation: Val → RC 1 → ST(RM) next PC → RSA next PC → REA Repeat a single instruction by the number in the RC (repeat count) register. Val is a 24-bit immediate value that is loaded into the RC (repeat counter) register. The RM (repeat mode) and RS (repeat single) status bits are set to 1, and the address of the next instruction is loaded into the repeat start address (RSA) and repeat end address (REA) registers.

Note: Optional syntaxes are shown in [*brackets*].

Key:

Src – General addressing modes
Src1 – Three-operand addressing modes
Src2 – Three-operand addressing modes
Csrc – Conditional branch addressing modes
Sreg – Register mode (any register)
Count – Shift value (general addressing modes)

Dreg – Register mode (any register)
Rn – Register mode (R0–R7)
Daddr – Destination memory address
ARn – Auxiliary register n (AR0–AR7)
Addr – 24-bit immediate address (label)
Cond – Condition code (see Table 6-2, pg. 6-4)

Syntax	Description
SIGI	**Signal, Interlocked** Operation: Signal interlocked operation Wait for interlock acknowledge Clear interlock An interlocked operation is signaled over XF0 and XF1. After the interlocked operation is acknowledged, it ends.
STF *Rn,Daddr*	**Store Floating-Point Value** Operation: Rn → Daddr Store the contents of an extended-precision register into a word in memory. The value that is stored is a floating-point number.
STFI *Rn,Daddr*	**Store Floating-Point Value, Interlocked** Operation: Rn → Daddr Signal end of interlocked operation The contents of the source operand are stored at the destination. An interlocked operation is signaled over XF0 and XF1. The operands are floating-point values.
STI *Sreg,Daddr*	**Store Integer** Operation: Sreg → Daddr Store the contents of the source register into a word in memory. The value that is stored is an integer.
STII *Sreg,Daddr*	**Store Integer, Interlocked** Operation: Sreg → Daddr Signal end of interlocked operation The contents of the source operand are stored at the destination. An interlocked operation is signaled over XF0 and XF1. The operands are signed integers.
SUBB *Src,Dreg*	**Subtract Integers with Borrow** Operation: Dreg - Src - C → Dreg Load the difference between the destination register, the source operand, and the carry bit into the destination register. The operands are signed integers.
SUBB3 *Src2,Src1,Dreg* [**SUBB** *Src2,Src1,Dreg*]	**Subtract Integers with Borrow (3-Operand)** Operation: Src1 - Src2 - C → Dreg Load the difference between the source operands and the carry bit into the destination register. The operands are signed integers.
SUBC *Src,Dreg*	**Subtract Integers Conditionally** Operation: If Dreg - Src ≥ 0 [(Dreg-Src) << 1] OR 1 →Dreg Else Dreg <<1 → Dreg If the difference between the destination and the source operands is greater than or equal to 0, then shift the difference left 1 bit, set the LSB to 1, and store the result in the destination register. If the difference between the destination and the source is less than zero, left shift the contents of the destination register by 1 bit. SUBC is equivalent to a single step of an integer divide. The operands are unsigned integers.

Syntax	Description
SUBF *Src,Rn*	**Subtract Floating-Point Values** Operation: Rn - Src → Rn Subtract the source operand from the contents of the extended-precision register and store the result in the register. Both operands are floating-point numbers.
SUBF3 *Src2,Src1,Rn* [**SUBF** *Src2,Src1,Rn*]	**Subtract Floating-Point Values (3-Operand)** Operation: Src1 - Src2 → Rn Subtract source 2 from source 1 and store the result in the extended-precision register. All the operands are floating-point numbers.
SUBI *Src,Dreg*	**Subtract Integers** Operation: Dreg - Src → Dreg Subtract the source operand from the contents of the destination register and store the result in the destination register. Both operands are signed integers.
SUBI3 *Src2,Src1,Dreg* [**SUBI** *Src2,Src1,Dreg*]	**Subtract Integers (3-Operand)** Operation: Src1 - Src2 → Dreg Subtract source 2 from source 1 and store the result in the destination register. All the operands are signed integers.
SUBRB *Src,Dreg*	**Subtract Reverse Integer with Borrow** Operation: Src - Dreg - C → Dreg Load the difference between the source, destination, and carry bit into the destination register. Both operands are signed integers.
SUBRF *Src,Rn*	**Subtract Reverse Floating-Point Value** Operation: Src - Rn → Rn Subtract the contents of the extended-precision register from the source operand and store the result into the register. Both operands are floating-point numbers.
SUBRI *Src,Dreg*	**Subtract Reverse Integer** Operation: Src - Dreg → Dreg Subtract the contents of the destination register from the source operand and store the result into the destination register. Both operands are signed integers.
SWI	**Software Interrupt** Operation: Perform emulator interrupt sequence.

Note: Optional syntaxes are shown in [*brackets*].

Key:

Src – General addressing modes
Src1 – Three-operand addressing modes
Src2 – Three-operand addressing modes
Csrc – Conditional branch addressing modes
Sreg – Register mode (any register)
Count – Shift value (general addressing modes)
Dreg – Register mode (any register)
Rn – Register mode (R0–R7)
Daddr – Destination memory address
ARn – Auxiliary register n (AR0–AR7)
Addr – 24-bit immediate address (label)
Cond – Condition code (see Table 6-2, pg. 6-4)

Syntax	Description
TRAP*cond N* [**TRAP** *N*]	**Trap Conditionally or Unconditionally** Operation: 0 → ST(GIE) If *cond*= true next PC → *++SP trap vector N → PC Else Set ST(GIE) to original state continue If the condition is true or missing, the PC contents are pushed on the system stack, the PC is loaded with the contents of the specified trap vector (*N*), and interrupts are disabled. *N* is an immediate value from 0–31.
TSTB *Src,Dreg*	**Test Bit Fields** Operation: Dreg AND Src Perform a bitwise logical AND of the source and destination and set the appropriate flags on the result. This is a nondestructive compare; the results of the compare are not stored. The source operand is an unsigned integer.
TSTB3 *Src1,Src2* [**TSTB** *Src1,Src2*]	**Test Bit Fields (3-Operand)** Operation: Src1 AND Src2 Perform a bitwise logical AND of the two source operands and set the appropriate flags on the result. This is a nondestructive compare; the results of the compare are not stored. The source operands are unsigned integers.
XOR *Src,Dreg*	**Bitwise Exclusive OR** Operation: Dreg XOR Src → Dreg Perform a bitwise exclusive OR of the source and destination operands and store the result in the destination register. The source operand is an unsigned integer.
XOR3 *Src2,Src1,Dreg* [**XOR** *Src2,Src1,Dreg*]	**Bitwise Exclusive OR (3-Operand)** Operation: Src1 XOR Src2 → Dreg Perform a bitwise exclusive OR of the two source operands and store the result in the destination register. The source operands are unsigned integers.

Note: Optional syntaxes are shown in [*brackets*].

Key:

Src – General addressing modes
Src1 – Three-operand addressing modes
Src2 – Three-operand addressing modes
Csrc – Conditional branch addressing modes
Sreg – Register mode (any register)
Count – Shift value (general addressing modes)
Dreg – Register mode (any register)
Rn – Register mode (R0–R7)
Daddr – Destination memory address
ARn – Auxiliary register n (AR0–AR7)
Val – Immediate value
Cond – Condition code (see Table 6-2, pg. 6-4)

6.2 Three-Operand Instructions

Most instructions have only two operands; however, several arithmetic and logical instructions have three-operand versions. Three-operand instructions allow the TMS320C30 to read two operands from memory or the register file in a single cycle.

- Two-operand instructions have a single source operand (or shift count) and a destination operand.
- Three-operand instructions may have two source operands (or one source operand and a count operand) and a destination operand. A source operand may be a memory word or a register. The destination of a three-operand instruction is always a register.

Table 6-3 lists the instructions that have three-operand versions.

Table 6-3. Summary Three-Operand Instructions

Instruction	Description	Instruction	Description
ADDC3	Add with carry	ADDF3	Add floating-point values
ADDI3	Add integers	AND3	Bitwise logical AND
ANDN3	Bitwise logical AND with complement	ASH3	Arithmetic shift
CMPF3	Compare floating-point values	CMPI3	Compare integers
LSH3	Logical shift	MPYF3	Multiply floating-point values
MPYI3	Multiply integers	OR3	Bitwise logical OR
SUBB3	Subtract integers with borrow	SUBF3	Subtract floating-point values
SUBI3	Subtract integers	TSTB3	Test bit fields
XOR3	Bitwise exclusive-OR		

Note:

You can omit the *3* for all three-operand instructions.

6.3 Parallel Instructions

Some of the TMS320C30 instructions can occur in pairs that will be executed in parallel. Table 6-4 lists the valid instruction pairs. These **parallel instructions** allow:

- Parallel loading of register,
- Parallel arithmetic operations, **and**
- Arithmetic or logical instructions that can be used in parallel with a store instruction.

Each instruction in a pair is entered as a separate source statement; the second instruction must be preceded by two vertical bars (| |). This example shows the syntax for parallel instructions:

```
label:    ADDI3  R0,*AR0,R1        ; Part 1 (label is optional)
||        STI    R4,*+AR11.        ; Part 2
```

Note that the first instruction in the pair may have a label, but the second instruction **cannot** have a label.

The assembler allows several relaxed syntax forms for parallel instructions:

- The vertical bars can be placed in column 1 or anywhere between column 1 and the mnemonic. Here is another example of valid syntax for parallel instructions:

```
label:     MPYI3  R0,*AR1,R0
        || ADDI3  *AR2,R1,R2
```

- The instructions in a parallel instruction pair may be specified in either order. For instance, the preceding example could also be specified as:

```
label:     ADDI3  *AR2,R1,R2
        || MPYI3  R0,*AR1,R0
```

- If one of the instructions in a pair uses a three-operand instruction, you can omit the *3* for that instruction.

```
   MPYI3  R0,*AR1,R0   can be               MPYI   R0,*AR1,R0
|| ADDI3  *AR2,R1,R2  written as         || ADDI   *AR2,R1,R2
```

- All commutative operations can be written in either order. For example,

```
ADDI  *AR0,R1,R2   can be written as   ADDI  R1,*AR0,R2
```

- The third operand of a three-operand instruction specifies a destination register. You can omit the third operand if it is the same as the second operand. This allows you to use three-operand instructions that look like two-operand instructions. For example,

```
ADDI3  *AR0,R2,R2        can be       ADDI  *AR0,R2
MPYI3  *AR1,R0,R0      written as     MPYI  *AR1,R0
```

- Instructions that can use the preceding two syntaxes include:

ADDC3	AND3	LSH3	OR3	SUBI3
ADDF3	ANDN3	MPYF3	SUBB3	XOR3
ADDI3	ASH3	MPYI3		

Note that all registers are read at the beginning of the execution cycle and loaded at the end of the execution cycle. If an instruction in a pair reads a register and another instruction writes to the same register, then the former instruction uses the contents of the register *before it is modified by the latter instruction.*

Table 6-4. Summary of Parallel Instructions

Parallel Arithmetic with Store Instructions	
Syntax	**Operation**
ABSF Src2,Dst1 || **STF** Src3,Dst2	|Src2| → Dst1 || Src3 → Dst2
ABSI Src2,Dst1 || **STI** Src3,Dst2	|Src2| → Dst1 || Src3 → Dst2
ADDF3 Src1,Src2,Dst1 || **STF** Src3,Dst2	Src1 + Src2 → Dst1 || Src3 → Dst2
ADDI3 Src1,Src2,Dst1 || **STI** Src3,Dst2	Src1 + Src2 → Dst1 || Src3 → Dst2
AND3 Src2,Src1,Dst1 || **STI** Src3,Dst2	Src1 AND Src2 → Dst1 || Src3 → Dst2
ASH3 Count,Src2,Dst1 || **STI** Src3,Dst2	If Count ≥ 0 Src2 << Count → Dst1 || Src3 → Dst2 Else Src2 >> |Count| → Dst1 || Src3 → Dst2
FIX Src2,Dst1 || **STI** Src3,Dst2	Fix(Src2) → Dst1 || Src3 → Dst2
FLOAT Src2,Dst1 || **STF** Src3,Dst2	Float(Src2) → Dst1 || Src3 → Dst2
LDF Src2,Dst1 || **STF** Src3,Dst2	Src2 → Dst1 || Src3 → Dst2
LDI Src2,Dst1 || **STI** Src3,Dst2	Src2 → Dst1 || Src3 → Dst2
LSH3 Count,Src2,Dst1 || **STI** Src3,Dst2	If Count ≥ 0 Src2 << Count → Dst1 || Src3 → Dst2 Else Src2 >> |Count| → Dst1 || Src3 → Dst2
MPYF3 Src2,Src1,Dst1 || **STF** Src3,Dst2	Src1 × Src2 → Dst1 || Src3 → Dst2

Key:
Src1 – Register mode (R0–R7)
Src3 – Register mode (R0–R7)
Dst1 – Register mode (R0–R7)
Op3 – Register mode (R0 or R1)
Src2 – Indirect mode (disp. = 0, 1, IR0, IR1)
Sr42 – Indirect mode (disp. = 0, 1, IR0, IR1)
Dst2 – Indirect mode (disp. = 0, 1, IR0, IR1)
Op6 – Register mode (R2 or R3)
Op1,Op2,Op4,Op5 – Two of these operands must be specified using register mode and two must be specified using indirect mode

Table 6-4. Summary of Parallel Instructions (Concluded)

Syntax	Operation
Parallel Arithmetic with Store Instructions (continued)	
Syntax	**Operation**
MPYI3 Src2,Src2,Dst1 || **STI** Src3,Dst2	Src1 × Src2 → Dst1 || Src3 → Dst2
NEGF Src2,Dst1 || **STF** Src3,Dst2	0 - Src2 → Dst1 || Src3 → Dst2
NEGI Src2,Dst1 || **STI** Src3,Dst21	0 - Src2 → Dst1 || Src3 → Dst2
NOT Src2,Dst1 || **STI** Src3,Dst2	$\overline{\text{Src2}}$ → Dst1 || Src3 → Dst2
OR3 Src2,Src1,Dst1 || **STI** Src3,Dst2	Src1 OR Src2 → Dst1 || Src3 → Dst2
STF Src1,Dst1 || **STF** Src3,Dst2	Src1 → Dst1 || Src3 → Dst2
STI Src1,Dst1 || **STI** Src3,Dst2	Src1 → Dst1 || Src3 → Dst2
SUBF3 Src2,Src1,Dst1 || **STF** Src3,Dst2	Src1 - Src2 → Dst1 || Src3 → Dst2
SUBI3 Src2,Src1,Dst1 || **STI** Src3,Dst2	Src1 - Src2 → Dst1 || Src3 → Dst2
XOR3 Src2,Src1,Dst1 || **STI** Src3,Dst2	Src1 XOR Src2 → Dst1 || Src3 → Dst2
Parallel Load Instructions	
Syntax	**Operation**
LDF Src2,Dst1 || **LDF** Src4,Dst2	Src2 → Dst1 || Src4 → Dst2
LDI Src2,Dst1 || **LDI** Src4,Dst2	Src2 → Dst1 || Src4 → Dst2
Parallel Multiply and Add/Subtract Instructions	
Syntax	**Operation**
MPYF3 Op1,Op2,Op3 || **ADDF3** Op4,Op5,Op6	Op1 × Op2 → Op3 || Op4 + Op5 → Op6
MPYF3 Op1,Op2,Op3 || **SUBF3** Op4,Op5,Op6	Op1 × Op2 → Op3 || Op4 - Op5 → Op6
MPYI3 Op1,Op2,Op3 || **ADDI3** Op4,Op5,Op6	Op1 × Op2 → Op3 || Op4 + Op5 → Op6
MPYI3 Op1,Op2,Op3 || **SUBI3** Op4,Op5,Op6	Op1 × Op2 → Op3 || Op4 - Op5 → Op6

Key:
Src1 – Register mode (R0–R7)
Src2 – Indirect mode (disp. = 0, 1, IR0, IR1)
Src3 – Register mode (R0–R7)
Sr42 – Indirect mode (disp. = 0, 1, IR0, IR1)
Dst1 – Register mode (R0–R7)
Dst2 – Indirect mode (disp. = 0, 1, IR0, IR1)
Op3 – Register mode (R0 or R1)
Op6 – Register mode (R2 or R3)
Op1,Op2,Op4,Op5 – Two of these operands must be specified using register mode and two must be specified using indirect mode

6.4 Load and Store Instructions

The TMS320C30 supports 12 load and store instructions, which are summarized in Table 6-5. These instructions:

- Load a word from memory into a register,
- Store a word from a register into memory, **or**
- Manipulate data on the system stack.

The TMS320C30 also provides you with the ability to load data conditionally; this is useful for locating the maximum or minimum value in a data set.

Table 6-5. Summary of Load and Store Instructions

Instruction	Description	Instruction	Description
LDE	Load floating-point exponent	POP	Pop integer from stack
LDF	Load floating-point value	POPF	Pop floating-point value from stack
LDF*cond*	Load floating-point value conditionally	PUSH	Push integer on stack
LDI	Load integer	PUSHF	Push floating-point value on stack
LDI*cond*	Load integer conditionally	STF	Store floating-point value
LDM	Load floating-point mantissa	STI	Store integer

6.5 Arithmetic Instructions

The TMS320C30 supports a complete set of arithmetic instructions. These instructions provide integer operations, floating-point operations, and multi-precision arithmetic. Table 6-6 summarizes these instructions.

Table 6-6. Summary of Arithmetic Instructions

Instruction	Description	Instruction	Description
ABSF	Absolute value of a floating-point number	NEGB	Negate integer with borrow
ABSI	Absolute value of an integer	NEGF	Negate floating-point value
ADDC †	Add integers with carry	NEGI	Negate integer
ADDF †	Add floating-point values	NORM	Normalize floating-point value
ADDI †	Add integers	RND	Round floating-point value
ASH †	Arithmetic shift	SUBB †	Subtract integers with borrow
CMPF †	Compare floating-point values	SUBC	Subtract integers conditionally
CMPI †	Compare integers	SUBF †	Subtract floating-point values
FIX	Convert floating-point value to integer	SUBRB	Subtract reverse-integer with borrow
FLOAT	Convert integer to floating-point value	SUBRF	Subtract reverse floating-point value
MPYF †	Multiply floating-point values	SUBRI	Subtract reverse integer
MPYI †	Multiply integers		

† Two and three operand versions

6.6 Logical Instructions

The TMS320C30 supports a complete set of logical instructions, which are summarized in Table 6-7.

Table 6-7. Summary of Logical Instructions

Instruction	Description	Instruction	Description
AND †	Bitwise logical AND	ROLC	Rotate left through carry
ANDN †	Bitwise logical AND with complement	ROR	Rotate right
LSH †	Logical shift	RORC	Rotate right through carry
NOT	Bitwise logical complement	TSTB †	Test bit fields
OR †	Bitwise logical OR	XOR †	Bitwise exclusive OR
ROL	Rotate left		

† Two and three operand versions

6.7 Program-Control Instructions

These instructions control program flow by providing repeat modes (zero-overhead looping) and branching. The repeat modes support repetition of a block of code or of a single line of code. Both standard and delayed branching are supported. Table 6-8 lists the program-control instructions.

Table 6-8. Summary of Program-Control Instructions

Instruction	Description	Instruction	Description
B*cond*[D]	Branch conditionally (standard or delayed)	NOP	No operation
BR[D]	Branch unconditionally standard or delayed)	RETI *cond*	Return from interrupt conditionally
CALL	Call subroutine	RETS *cond*	Return from subroutine conditionally
CALL*cond*	Call subroutine conditionally	RPTB	Repeat block of instructions
DB*cond*[D]	Decrement and branch conditionally	RPTS	Repeat single instruction
IDLE	Idle until interrupt	TRAP *cond*	Trap conditionally
SWI	Software interrupt		

6.8 Interlocked-Operation Instructions

The interlocked-operations instructions support multiprocessor communication. Table 6-9 lists the interlocked-operation instructions.

Table 6-9. Summary of Interlocked-Operation Instructions

Instruction	Description	Instruction	Description
LDFI	Load floating-point value, interlocked	STFI	Store floating-point value, interlocked
LDII	Load integer, interlocked	STII	Store integer, interlocked
SIGI	Signal, interlocked		

6.9 The LDP Instruction

The LDP (load data page) instruction is a special form of the LDI (load integer) instruction. LDP allows you to load a register (usually the DP register) with the page number of a relocatable address. A page number is represented by the eight MSBs of a 24-bit address. The page number is combined with the 16 LSBs of an instruction word to form a direct address.

The syntax for the LDP instruction is:

[*label*] **LDP** *expression*[*,register*]

LDP assembles as an LDI instruction with an immediate source operand.

- The *expression* is a relocatable address, which is usually represented by a symbol name.
- The 8 MSBs of the address are loaded into the destination *register*. If you do not specify a *register*, the assembler will use the DP register as a default.

At link time, *expression* may be relocated to a different page than it occupied at assembly time. The assembler generates a special relocation type that allows the linker to patch the correct page number into the LDP instruction.

The following example illustrates use of the LDP instruction. Assume a variable named `sym` is defined in the .bss section as shown:

```
.bss  sym,1     ; Allocate sym in .bss
```

To read the value of `sym` using direct addressing, you must first load the DP register with the 8-bit pointer to the page on which `sym` is located. Normally, you do not know at assembly time where the .bss section will be loaded, so you must use an LDP instruction to load DP before accessing the variable:

```
LDP   sym       ; Load DP with page number of sym
LDI   @sym, R0  ; Use direct addressing to access sym
```

Note that the *register* operand was omitted from the LDP instruction; DP was used as the default.

Section 7

Macro Language

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

- Define your own macros
- Redefine existing opcodes and macros
- Access macro libraries created with the archiver
- Manipulate strings within a macro
- Define conditional and repeatable blocks within a macro
- Control macro expansion listing

There are three phases of macro use:

- **Macro definition**. Macros must be defined before they can be invoked. There are two methods for defining macros:

  1) Macros can be defined in the **source file** where they are used (or in a separate text file that is included with a .copy or .include directive). Because macros must be defined before they are called, it is a good practice to place all the definitions at the beginning of the file.

  2) 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 that corresponds to the name of the member. You can access a macro library by using the .mlib directive. Because macros must be defined before they can be called, the .mlib directive must appear in the source code before any of the macros in the library are called.

- **Macro invocation**. Once a macro has been defined, the macro name can be used as an opcode in a source program. This is referred to as a *macro call*.
- **Macro expansion**. When the source program calls a macro, the assembler substitutes the statements within the macro definition for the macro call statement.

This section discusses the following topics:

Section **Page**

7.1 Macro Directives Summary 7-2
7.2 Macro Libraries 7-3
7.3 Defining Macros 7-4
7.4 Macro Parameters 7-6
7.5 Conditional Blocks 7-7
7.6 Repeatable Blocks 7-8
7.7 Unique Labels 7-9

7.1 Macro Directives Summary

Directive	Description
$MACRO	*Macro Definition Directive* **Syntax:** *macro name* **$MACRO** *[parm$_1$[, ... , parm$_n$]]* The $MACRO directive begins a macro definition. It must be the first statement in a macro definition. $MACRO assigns a name to the macro and declares the macro parameters. *macro name* is the name of the macro. A macro name may be 1 to 32 alphanumeric characters; it must begin with a letter. *Parms* are optional parameters. When a macro is called, the assembler will associate the first operand in the macro call with the first parameter of the macro definition, and so on.
$IF	*Begin Conditional Block Directive* **Syntax:** **$IF** *expression* The $IF directive begins a conditional block. If the *expression* evaluates to a non-zero value, then the code following the $IF directive (up to an $ELSE or $ENDIF directive) will be assembled.
$ELSE	*Alternate Conditional Block Directive* **Syntax:** $ELSE The $ELSE directive may be used within a conditional block. If the *expression* in an $IF directive evaluates to 0, then code following a corresponding $ELSE directive (up to an $ENDIF directive) will be assembled.
$ENDIF	*Terminate Conditional Block Directive* **Syntax:** $ENDIF The $ENDIF directive terminates a conditional block.
$ENDM	*Terminate Macro Definition Directive* **Syntax:** $ENDM The $ENDM directive terminates a macro definition.
$LOOP	*Begin Repeatable Block Directive* **Syntax:** **$LOOP** *expression* The $LOOP directive begins a repeatable block. The expression is evaluated only once; the expression should evaluate to a value in the range 0–32767.
$ENDLOOP	*Terminate Repeatable Block Directive* **Syntax:** $ENDLOOP The $ENDLOOP directive terminates a repeatable block.

7.2 Macro Libraries

A macro library is a collection of files that contain macro definitions. These files, or members, are bound into a single file (called an archive) by the archiver. Each member of a macro library may contain one macro definition. The macro name and the member name must be the same, and the macro filename's extension must be .asm. The files in a macro library must be unassembled source files. You can access the macro library by using the .mlib assembler directive:

.mlib *"macro library filename"*

When the assembler encounters an .mlib directive, it opens the library and creates a table of its contents. The assembler enters the names of the individual members within the library 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 manner as 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.

You can create a macro library with the archiver by simply 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 following example creates a macro library called `maclib.lib`:

```
ar30 -a maclib.lib mac1.asm mac2.asm mac3.asm mac4.asm
```

This example adds four macro files (`mac1.asm`, `mac2.asm`, `mac3.asm`, and `mac4.asm`) to the library `maclib.lib`. Note that this could be a new or an existing library; if the library already existed, this example would simply append the macros to the end of the library.

Now you can specify `maclib.lib` to the assembler with an .mlib directive, and call any of the macros that it contains:

```
        .mlib   "maclib.lib"
        mac1                    ; Macro call
```

The assembler assumes that the files in the archive contain macro definitions with the same names as the members. The assembler expects **only** macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable effects.

7.3 Defining Macros

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

macname **$MACRO** *[parm$_1$] [,parm$_2$] ... [,parm$_n$]*
"
"
"
model statements or *macro directives*
"
"
"
$ENDM

where:

macname names the macro. It must be placed in the source statement's label field. Macro names are significant to 32 characters. The assembler places this name in the internal opcode table, replacing any instruction or previous macro definition with the same name.

$MACRO identifies this source statement as the first line of a macro definition; it must be placed in the opcode field.

parms are optional parameters which may appear as operands for the $MACRO directive. Parameters are not required by all macros.

model statements
are instructions or assembler directives that are used each time the macro is invoked.

macro directives
control the expansion of the macro or manipulate macro parameters.

$ENDM terminates the macro definition.

The contents of a single macro definition must be contained in the same file. Macro definitions cannot be nested, but other directives, instructions, and macro calls can be used in a macro definition. The assembler performs only limited error checking of macro definitions (during the definition phase), so multiple expansions of a macro may produce duplicate error messages.

When a macro is called, the assembler substitutes the model statements and macro directives within the definition for the macro call in the source code. Example 7-1 shows an example of a macro definition, how it is called, and how it is expanded in the source code.

Example 7-1. Macro Definition, Call, and Expansion

Macro Definition: The following code defines a macro, MOVREG, that has three parameters.

```
0001            *******************************************************
0002            MOVREG   $MACRO     p1,p2,pN     ; Begin macro definition
0003                     LDI        :p1:,:p2:    ; Model statement
0004                     LDI        :p2:,:pN:    ; Model statement
0005                     $LOOP      2            ; Begin repeat block
0006                     NOP                     ; Model statement
0007                     $ENDLOOP                ; End repeat block
0008                     $ENDM                   ; End macro definition
```

Macro Call: The MOVREG macro is invoked in the source code.

```
0009            *******************************************************
0010                     MOVREG     R0,R1,R2     ; Macro call
```

Macro Expansion: The assembler substitutes the functional lines of the macro definition for the macro call. The macro parameters are replaced with the operands supplied in the macro call.

```
!0001 000000 08010000    LDI        R0,R1
!0002 000001 08020001    LDI        R1,R2
!0003 000002 0C800000    NOP
!0004 000003 0C800000    NOP
```

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 encountered macro. This allows you to expand the functions of directives and instructions, as well as to add new instructions.

Caution:

When you specify a macro library with the .mlib directive, the assembler places all the entries in the specified library into the opcode table - not just the macros that are called. Make sure that the macros and instructions you want to use are not redefined by macros in a macro library.

7.4 Macro Parameters

Macros can declare local parameters whose scope is limited to the defining macro. These parameters do not conflict with symbols defined outside the macro. Only the first eight characters of a parameter name are significant. A single macro can declare a maximum of 128 parameters.

The assembler assigns initial values to macro parameters when the macro is called. For example, consider the following macro definition line:

```
ADDUP  $MACRO   val1,val2,sum
```

This example defines three parameters (val1, val2, and sum). The assembler assigns values to these parameters when it expands the macro; each parameter corresponds to an operand in the macro call.

The value that is assigned to a macro parameter is called a **string value**. The assembler will substitute a parameter's string value into a model statement when you enclose the parameter name in colons. Parameters can be used this way anywhere in a model statement (as a label, an operand, etc.).

Example 7-2 shows a macro that has four parameters.

Example 7-2. Using Parameter Values

```
 0001                      packword   $MACRO     b1,b2,b3,b4
 0002                      * Make sure these are all in one word
 0003                                 .even
 0004                                 .field     :b1:,8
 0005                                 .field     :b2:,8
 0006                                 .field     :b3:,8
 0007                                 .field     :b4:,8
 0008                                 $ENDM
 0009
 0010        00000003      A          .set       03h
 0011        00000010      B          .set       10h
 0012        00000009      C          .set       09h
 0013        00000044      D          .set       44h
 0014
 0015 000000 00000037                 .field     37h,12
 0016
 0017                                 packword   A,B,C,D
!0001 000001                          .even
!0002 000001 00000003                 .field     A,8
!0003 000001 00001003                 .field     B,8
!0004 000001 00091003                 .field     C,8
!0005 000001 44091003                 .field     D,8
```

The packword macro packs four values into the four bytes of a word. The parameters b1, b2, b3, and b4 are assigned values corresponding to the values that are passed when the macro is called.

7.5 Conditional Blocks

The $IF, $ELSE, and $ENDIF directives are used to construct conditional blocks within macro definitions. Conditional blocks can be nested up to ten levels deep. Blocks at all nesting levels must always be terminated with an $ENDIF. The general format of a conditional block is:

$IF *well-defined expression*
    code to assemble if expression is true (≠ 0)
$ELSE
    code to assemble if expression is false (=0)
$ENDIF

If the expression in the $IF statement evaluates to a nonzero value (*true*), then the code that follows it (up to an $ELSE or $ENDIF) will be assembled. If the expression evaluates to 0 (*false*), then the assembler does not assemble the code that follows the $IF statement; if an $ELSE directive is present, the assembler assembles the code that follows it (up to the $ENDIF).

All directives ($IF, $ELSE, and $ENDIF) in a single conditional block *must appear in the same source module*; the $ENDIF cannot appear in an included file. A conditional block not terminated by the end of a source file (or upon encountering an $ENDM directive) will produce an error.

Conditional assembly directives that appear in a macro definition are evaluated each time the macro is expanded, not as it is defined. Unassembled code (code following a false $IF or an unused $ELSE) is not scanned; no copy/include files are opened and no macros are defined in such blocks.

Figure 7-1 shows an example of a macro with a conditional block.

```
 0001                      CMPR   $MACRO  p1,p2
 0002                             $IF     :p1: <> :p2:
 0003                             .string "not equal"
 0004                             $ELSE
 0005                             .string "equal"
 0006                             $ENDIF
 0007                             $ENDM
 0008
 0009  00000001            sym1      .set    1
 0010  00000002            sym2      .set    2
 0011
 0012  000000                         CMPR    sym1, sym2
!0001  000000 20746F6E                .string "not equal"
       000001 61757165
       000002 0000006C
```

Figure 7-1. An Example of a Conditional Block

7.6 Repeatable Blocks

Repeatable blocks allow a section of code (or a section of a macro definition) to be repeatedly expanded. This is particularly useful for table generation. The format of a repeatable block is:

$LOOP *well-defined expressions*

model statements or *macro directives*

$ENDLOOP

The assembler evaluates the expression once when it enters the loop, and then it repeats the block *expression* number of times. The expression may be any legal expression or macro expression.

The restrictions that apply to conditional blocks also apply to repeatable blocks. You can nest up to 10 blocks; you can nest conditional blocks within repeatable blocks, and repeatable blocks within conditional blocks. The assembler checks to see if blocks are nested properly; if they are not, the assembler produces an error message. The following example shows improper nesting:

```
$LOOP     expression 1
   .
   .
$IF       expression 2
$ENDLOOP
   .
   .
$ENDIF
```

Note that the two blocks overlap rather than nest properly. This is an error, and the macro definition will be ignored.

Example 7-3 shows an example of a repeatable block.

Example 7-3. A Repeatable Block

```
 0001                        fill    $MACRO     f_val
 0002                                $LOOP      32
 0003                                .word      :f_val:
 0004                                $ENDLOOP
 0005                                $ENDM
 0006
 0007                                fill       0AABBCCDDh
!0001 000000 AABBCCDD                .word      0AABBCCDDh
!0002 000001 AABBCCDD                .word      0AABBCCDDh
!0003 000002 AABBCCDD                .word      0AABBCCDDh
  .     .       .                      .            .
  .     .       .                      .            .
  .     .       .                      .            .
!0030 00001D AABBCCDD                .word      0AABBCCDDh
!0031 00001E AABBCCDD                .word      0AABBCCDDh
!0032 00001F AABBCCDD                .word      0AABBCCDDh
```

7.7 Unique Labels

Labels must be unique. If you use an ordinary label in a macro, and the macro is expanded more than once, the label in the macro defines the label/symbol more than once - *this is illegal.* The macro language supports a special form of label that allows you to create unique labels within macros. To form a unique label, *simply follow the label name with a question mark*; the syntax for a unique label is:

label?

Symbols that are defined in this manner can be used like any other symbol; you can declare them as global symbols, you can use them in expressions, etc.

Section 8

Archiver Description

The TMS320C30 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 file, you can use the archiver to add more files to it, delete or replace existing members, or extract members.

You can build libraries out of 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 to build a library of object modules. For example, you could write several arithmetic routines, assemble them, and then 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 will search through the library and include any members that resolve external references.

You can also use the archiver to build macro libraries. You can create several separate source files, each of which contains a single macro, and then use the archiver to collect these macros into a single, functional group. The .mlib assembler directive lets you specify the name of a macro library to the assembler; during the assembly process, the assembler will search the specified library for the macros that you call. Section 7 discusses macros and macro libraries in detail.

This section contains the following topics:

Section **Page**

8.1 Archiver Development Flow 8-2
8.2 Invoking the Archiver 8-3
8.3 Archiver Examples 8-4

8.1 Archiver Development Flow

Figure 8-1 shows the archiver's role in the assembly language development process. Both the assembler and the linker accept libraries as input.

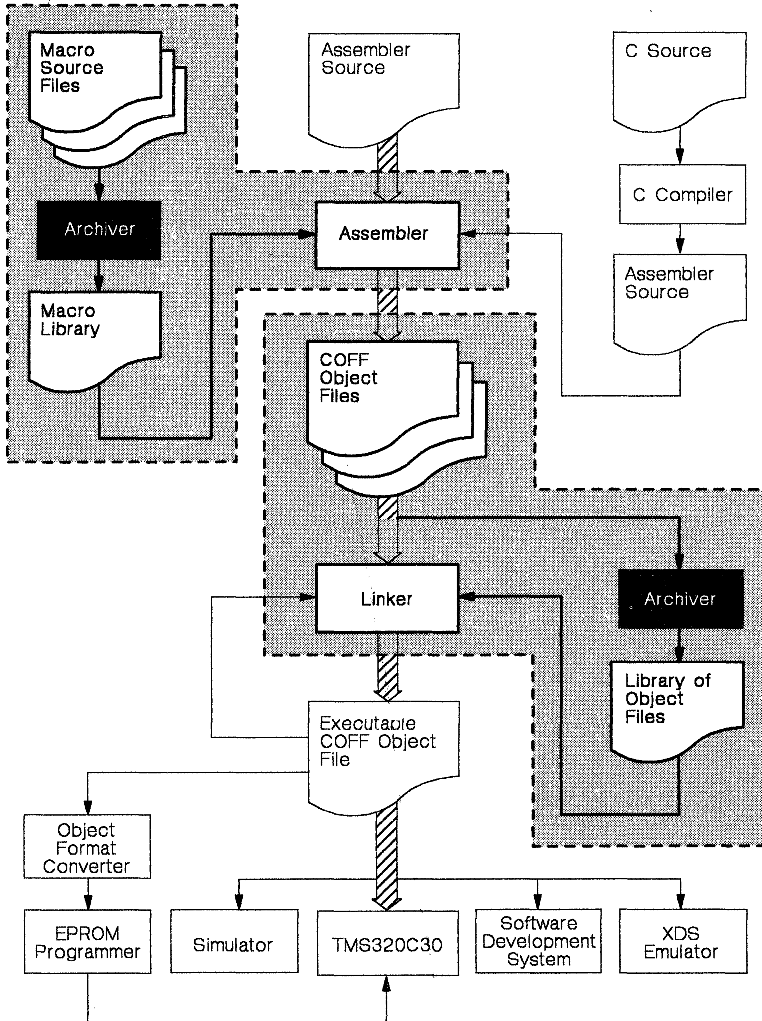


Figure 8-1. Archiver Development Flow

8.2 Invoking the Archiver

To invoke the archiver, enter:

ar30 *[-]command[option] libname [filename$_1$ ... filename$_n$]*

ar30 is the command that invokes the archiver; *libname* names an archive library. If you don't specify an extension for *libname*, the archiver uses the default extension ***.lib***. The *filenames* name individual member files that are associated with the library. If you don't specify an extension for a *filename*, the archiver uses the default extension ***.obj***.

The *command* tells the archiver how to manipulate the members in the library. 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 only use **one** command per invocation. Valid archiver commands include:

a adds the specified files to the library. Note that 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. It is possible for an archive to contain several members that have the same name. If you want to *replace* existing members, use the **r** command.

d deletes the specified members from the library.

r replaces the specified members in the library. If you don't specify any 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 don't specify any filenames, the archiver lists all the members in the specified library.

x extracts the specified files. If you don't specify any member names, the archiver extracts all the members in the library. When the archiver extracts a member, it simply copies the member into the current directory; it *doesn't* remove it from the library.

In addition to one of the *commands*, you can specify the following *options*:

e tells the archiver not to use the default extension .obj for member names.

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.)

v (verbose) provides a file-by-file description of the creation of a new library from an old library and its constituent members.

Note:

It is possible (but not desirable) for a library to contain several members with the same name. If you attempt to delete, replace, or extract a member, and the library contains more than one member with the specified name, then the archiver deletes, replaces, or extracts the *first* member with that name.

8.3 Archiver Examples

Here are some examples of using the archiver.

- **Example 1:**

  This example creates a library called `function.lib` that contains the files `sine.obj`, `cos.obj`, and `flt.obj`.

```
ar30 -a function sine cos flt
TMS320C30 Archiver   Version 1.10.01
(c) Copyright 1987, 1988, Texas Instruments Inc.
  ==>  new archive 'function.lib'
  ==>  building archive 'function.lib'
```

  Since these examples use the default extensions (`.lib` for the library and `.obj` for the members), it is not necessary to specify them.

- **Example 2:**

  You can print a table of contents of `function.lib` with the -t option:

```
ar30 -t function
TMS320C30 Archiver            Version 5.xx 87.160
(c) Copyright 1987, Texas Instruments Inc.
         FILE NAME    SIZE   DATE
 ----------------    -----   ------------------------
          sine.obj     248   Mon Nov 19 01:25:44 1984
           cos.obj     248   Mon Nov 19 01:25:44 1984
           flt.obj     248   Mon Nov 19 01:25:44 1984
```

- **Example 3:**

  You can explicitly specify extensions if you don't want the archiver to use the default extensions; for example:

```
ar30 -ave function.fn sine.asm cos .asm flt.asm
TMS320C30 Archiver   Version 1.10.01
(c) Copyright 1987, 1988, Texas Instruments Inc.
  ==>  add 'sine.asm'
  ==>  add 'cos.asm'
  ==>  add 'flt.asm'
  ==>  building archive 'function.fn'
```

  This creates a library called `function.fn` that contains the files `sine.asm`, `cos.asm`, and `flt.asm`. (-v is the verbose option.)

- **Example 4:**

  If you wanted to add some new members to a library, specify:

```
ar30 -as function tan.obj arctan.obj area.obj
TMS320C30 Archiver  Version 1.10.01
(c) Copyright 1987, 1988, Texas Instruments Inc.
  ==>  symbol defined: 'K2'
  ==>  symbol defined: 'Rossignol'
  ==>  building archive 'function.lib'
```

```
ar30 -a function tan.obj arctan.obj area.obj
```

Since this example doesn't specify an extension for the libname, the archiver adds the files to the library called function.lib. If function.lib didn't exist, the archiver would create it. (The -s option tells the archiver to list the global symbols that are defined in the library.)

- **Example 5:**

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

  ```
  ar30 -x macros push.asm
  ```

  The archiver makes a copy of push.asm and places it in the current directory; it doesn't remove push.asm from the library, though. Now you can edit the extracted file. To replace the copy of push.asm that's in the library with the copy that was changed, enter:

  ```
  ar30 -r macros push.asm
  ```

Section 9

Linker Description

The TMS320C30 linker creates executable modules by combining COFF object files. The concept of COFF *sections* is basic to linker operation; Section 3 discusses COFF sections in detail.

As the linker combines object files, it performs the following tasks:

- It allocates sections into the target system's configured memory.
- It relocates symbols and sections to assign them to final addresses.
- It resolves undefined external references between input files.

The linker supports a C-like command language that controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation, and provides two powerful directives, MEMORY and SECTIONS, that allow you to:

- Define a memory model that conforms to target system memory,
- Combine object file sections,
- Allocate sections into specific areas of memory, **and**
- Define or redefine global symbols at link time.

Topics in this section include:

Section **Page**

9.1 Linker Development Flow 9-2
9.2 Invoking the Linker 9-3
9.3 Linker Options 9-4
9.4 Linker Command Files 9-11
9.5 Object Libraries 9-13
9.6 The MEMORY Directive 9-14
9.7 The SECTIONS Directive 9-16
9.8 Overlay Pages 9-23
9.9 Default Allocation 9-27
9.10 Special Section Types (DSECT, COPY, and NOLOAD) 9-29
9.11 Assigning Symbols at Link Time 9-30
9.12 Creating and Filling Holes 9-33
9.13 Partial (Incremental) Linking 9-37
9.14 Linking C Code 9-38
9.15 Linker Example 9-41

9.1 Linker Development Flow

Figure 9-1 illustrates the linker's role in the assembly language 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 TMS320C30.

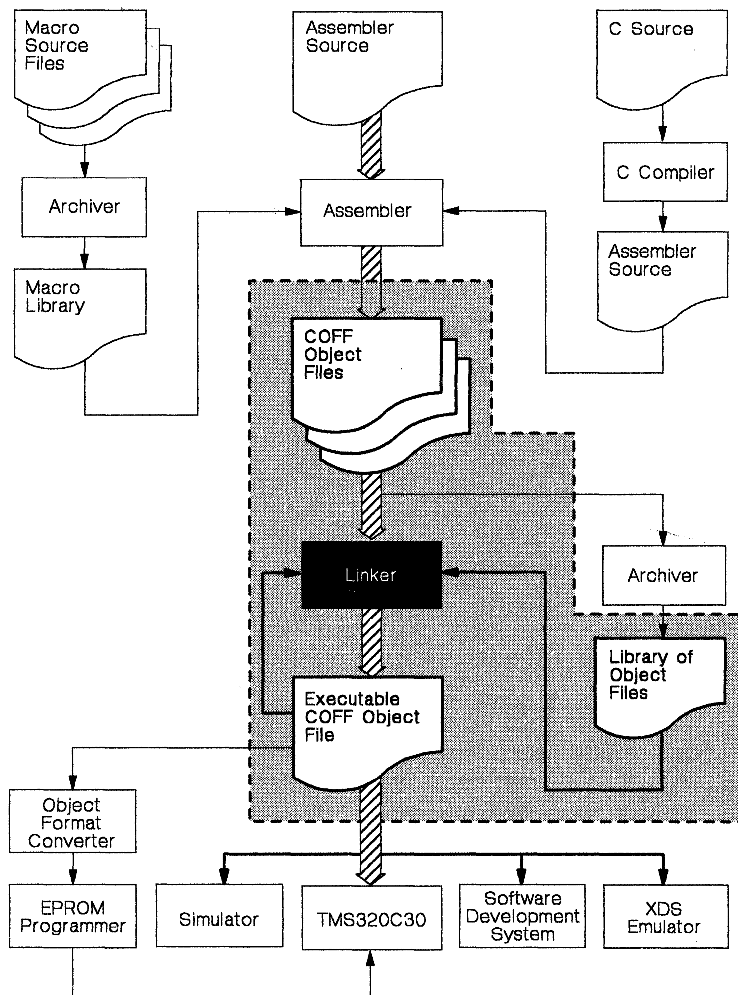


Figure 9-1. Linker Development Flow

9.2 Invoking the Linker

The general syntax for invoking the linker is:

lnk30 *[-options] filename$_1$ ... filename$_n$*

lnk30 is the command that invokes the linker. The *options* (discussed in Section 9.3) can appear anywhere on the command line or in a linker command file. The *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 file or an ASCII file that contains linker commands. The default output filename is ***a.out***.

There are three methods for invoking the linker:

- Specify options and filenames on the command line. This example links two files, `file1.obj` and `file2.obj`, and uses the -o option to create an output module named `link.out`.

  ```
  lnk30 file1.obj file2.obj -o link.out
  ```

- Enter the **lnk30** command with no filenames and no options; the linker will prompt for them:

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

  For *command files*, enter one or more command file names.

  For *object files*, enter one or more object file names. The default extension is *obj*. Separate the filenames with spaces or commas; if the last character is a comma, the linker will prompt for an additional line of object file names.

  The *output file* is the name of the linker output module. This overrides any -o options entered with any of the other prompts. If there are no -o options and you do not answer this prompt, the linker will create an object file with the default filename of ***a.out***.

  The *options* prompt is for additional options, although you can also enter options in a command file. Enter them with hyphens, just as you would on the command line.

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

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

  Now you can invoke the linker from the command line; specify the command file name as an input file: **`lnk30 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:
  `lnk30 -m link.map linker.cmd file3.obj`

  The linker reads and processes a command file as soon as it encounters it 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`.

9.3 Linker Options

Linker options control linking operations. They can be placed on the command line or in a command file. All linker options must be preceded by a hyphen (-). The order in which options are specified is unimportant, except for the -l and -i options. Options are separated from arguments (if they have them) by an optional space. Table 9-1 summarizes the linker options.

Table 9-1. Linker Options Summary

Option	Description
-a	Produce an absolute, executable module. This is the default; if neither -a nor -r is specified, the linker acts as if -a is specified.
-ar	Produce a relocatable, executable object module.
-c	Use linking conventions defined by the ROM autoinitialization model of the C compiler.
-cr	Use linking conventions defined by the RAM autoinitialization model of the C compiler.
-e	Defines a *global symbol* that specifies the primary entry point for the output module.
-f *fill value*	Set the default fill value for holes within output sections; *fill value* is a 4-byte constant.
-h	Make all global symbols static.
-i *dir*	Alter the library-search algorithm to look in *dir* before looking in the default location. This option must appear before the -l option.
-l *filename*†	Name an archive library file as linker input; *filename* is an archive library name.
-m *filename*†	Produce a map or listing of the input and output sections, including holes, and place the listing in *filename*.
-o *filename*†	Name the executable output module. The default filename is *a.out*.
-q	Request a quiet run (suppress the banner).
-r	Retain relocation entries in the output module.
-s	Strip symbol table information and line number entries from the output modules.
-u *symbol*	Place an unresolved external *symbol* into the output module's symbol table.

† The *filename* must follow operating system conventions.

9.3.1 Relocation Capability (-a and -r Options)

One of the tasks the linker performs is *relocation*. Relocation is the process of adjusting all the references to a symbol when the symbol's address changes. The linker supports two options (**-a** and **-r**) that allow you to choose whether you will produce an absolute or a 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 entries. Executable files:

  - Contain special symbols defined by the linker (Section 9.11.4, page 9-32, describes these symbols),
  - Contain an optional header that describes information such as the program entry point, **and**
  - Contain *no* unresolved references.

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

```
lnk30 -a file1.obj file2.obj
```

> **Note:**
>
> 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 an *unexecutable* file 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`:

  ```
  lnk30 -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 9.13, page 9-37.)

- Producing an *Executable* Relocatable Output Module (-ar)

  If you invoke the linker with both the -a and -r options, the linker produces an *executable, relocatable* object module. The output file contains special linker symbols, contains an optional header, and all symbol references are resolved (this is normal for a relocatable file); however, the relocation information is retained.

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

  ```
  lnk30 -ar file1.obj file2.obj -o xr.out
  ```

  Note that you can string the options together (lnk30 -ar) or you can enter them separately (lnk30 -a -r).

- Relocating or Relinking an Absolute Output Module

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

9.3.2 C Language Options (-c and -cr Options)

The -c and -cr options cause the linker to use linking conventions that are required by the TMS320C30 C compiler.

- The **-c** option tells the linker to use the ROM autoinitialization model.
- The **-cr** option tells the linker to use the RAM autoinitialization model.

For more information about linking C code, see section Section 9.14 on page 9-38.

9.3.3 Define an Entry Point (-e *global symbol* Option)

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

The linker can assign one of four possible 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. Possible entry point values include:

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 eternal symbol in one of the input files to be linked.
2) The value of symbol `_c_int00` (if present). `_c_int00` **must** be the entry point if you are linking code produced by the C compiler.
3) The value of symbol `_main` (if present).
4) Zero (default value).

This example links `file1.obj` and `file2.obj` and sets the entry point to the value of the symbol `begin`. This symbol must be defined as external in `file1` or `file2`.

```
lnk30 -e begin file1.obj file2.obj
```

9.3.4 Set Default Fill Value (-f *cc* 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 *cc* is a 4-byte constant (up to eight 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 AABBCCDDh:

```
lnk30 -f 0AABBCCDDh file1.obj file2.obj
```

9.3.5 Make All Global Symbols Static (-h Option)

The -h option makes all global symbols static. This "hides" symbols, because static symbols are not visible to externally linked modules. 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 were 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, these files can be linked without conflict. The symbol `ext` defined in `file1.obj` is treated separately from the symbol `ext` defined in `file2.obj`.

```
lnk30 -h file1.obj file2.obj
```

9.3.6 Alter the Library Search Algorithm (-i *dir* & -l *filename*/C—DIR)

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

```
lnk30 file1.obj object.lib
```

If you want to use a library 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; 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 in the following order:

1) It searches directories named with the -i linker option.
2) It searches directories named with the environment variable C—DIR.
3) If C—DIR is not set, it searches directories named with the assembler's environment variable, A—DIR.
4) It searches the current directory.

9.3.6.1 -i Linker Option

The -i option names an alternate directory that contains object libraries. The syntax for this option is **-i** *dir*. *dir* names a directory that contains object libraries; the space between -i and the directory name is optional. When the linker is searching for object libraries named with the -l option, it searches through directories named with -i first. Each -i option specifies only one directory, but you can use several -i options per invocation. When you use the -i option to name an alternate directory, it must precede the -l option on the command line or in a command file.

As an example, assume that two archive libraries called `r.lib` and `lib2.lib` reside in directories called:

- `\ld` and `\ld2` (DOS)
- `[ld]` and `[ld2]` (VMS), **or**
- `/ld` and `/ld2` (UNIX).

You can use both libraries during a link:

DOS: `lnk30 f1.obj f2.obj -i\ld -i\ld2 -lr.lib -llib2.lib`

VMS: `lnk30 f1.obj f2.obj -i[ld] -i[id2] -lr.lib -llib2.lib`

UNIX: `lnk30 f1.obj f2.obj -i/ld -i/ld2 -lr.lib -llib2.lib`

9.3.6.2 Environment Variable (C_DIR)

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 for assigning the environment variable is:

DOS: **set C_DIR**=*pathname; another pathname ...*

VMS: **assign C_DIR**"*pathname, another pathname...* "

UNIX: **setenv C_DIR**"*pathname; another pathname ...* "

The *pathnames* are directories that contain object libraries. Use the -l option on the command line or in a command file to tell the linker which libraries to search for.

As an example, assume that two archive libraries called `r.lib` and `lib2.lib` reside in directories called:

- `\ldir` and `\ldir2` (DOS),
- `[ldir]` and `[ldir2]` (VMS), **or**
- `/ldir` and `/ld2` (UNIX).

You can use both libraries during a link; set the environment variable first:

DOS:
```
set    C_DIR=\ldir;\ldir2
lnk30 f1.obj f2.obj -l r.lib -l lib2.lib
```

VMS:
```
assign C_DIR "[ldir];[ldir2]"
lnk30 f1.obj f2.obj -lr.lib -l lib2.lib
```

UNIX:
```
setenv C_DIR "/ldir;/ldir2
lnk30 f1.obj f2.obj  -l r.lib -l lib2.lib
```

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

DOS: `set C_DIR=`

VMS: `deassign C_DIR`

UNIX: `setenv 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 will search for object libraries in the directories named with A_DIR.

Section 9.5 (page 9-13) contains more information about object libraries.

9.3.7 Create a Map File (-m *filename* Option)

The -m option creates a link map listing and puts it in *filename*. This map describes:

- Memory configuration,
- Input and output section allocation, and
- The addresses of external symbols after they have been relocated.

The map file contains the name of the output module, the entry point, and may also contain up to three tables:

- A table showing the new memory configuration, if any nondefault memory is specified.
- A table showing the linked addresses of each output section and the input sections that make up the output sections.
- A table showing each external symbol and its address. This table has two columns: the left column contains the symbols sorted by name and the right column contains the symbols sorted by address.

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

```
lnk30 file1.obj file2.obj -m map.out
```

Section 9.15 (page 9-41) shows an example of a map file.

9.3.8 Name an Output Module (-o *filename* Option)

The linker always creates an executable output module. 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 have another name, use the -o option. The *filename* is the new output module name.

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

```
lnk30 -o run.out file1.obj file2.obj
```

9.3.9 Specify a Quiet Run (-q Option)

The -q option suppresses the linker's banner when -q is the first option on the command line or in a command file. This option is useful for batch operation.

9.3.10 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 places the output module, stripped of line numbers and symbol table information, named `nolink.out`:

```
lnk30 -o nolink.out -s file1.obj file2.obj
```

Note that using the -s option limits later use of a symbolic debugger, and may prevent a file from being relinked.

9.3.11 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 through a library and include the module that defines the symbol. Note that the linker must encounter the -u option *before* it links in the member that defines the symbol.

For example, suppose a library named `rts.lib` contains a member that defines the symbol `symtab`, none of the object files you are linking reference to `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 below forces the linker to search `rts.lib` for the member that defines `symtab` and to link in the member.

```
lnk30 -u symtab file1.obj file2.obj rts.lib
```

If you did not use -u, this member would not be included because there is no explicit reference to it in `file1.obj` or `file2.obj`.

9.4 Linker Command Files

Linker command files allow you to put linking information in a file; this is useful when you often invoke the linker 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 command line. Command files are ASCII files that contain one or more of the following:

- Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from the 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 allows you to specify the target memory configuration. The SECTIONS directive controls how sections are built and allocated.
- Assignment statements, which define and assign values to global symbols.

To invoke the linker with a command file, enter the **lnk30** command and follow it with the name of the command file:

lnk30 *command file name*

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links the file. Otherwise, it assumes a file is a command file and begins reading and processing commands from it.

Figure 9-2 shows a sample linker command file called `link.cmd`. (Figure 9-12 on page 9-42 contains another example of a linker command file.)

```
/*******************************************************/
/*        Sample 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    */
```

Figure 9-2. An example of a Linker Command File

This sample file in Figure 9-2 contains only filenames and options. (Note that you can place comments in a command file by delimiting them with /* and */.) To invoke the linker with this command file, enter:

```
lnk30 link.cmd
```

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

```
lnk30 -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 also 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 can enter:

```
lnk30 names.lst dir.cmd
```

A command file can call another command file; this type of nesting is limited to 16 levels. If a command file names another command file as input, this statement must be the *last* statement in the calling command file.

Blanks and blank lines that appear in a command file are insignificant except as delimiters. This also applies to the format of linker directives in a command file. Figure 9-3 shows a sample command file that contains linker directives. (Linker directive formats are discussed in later sections.)

```
/*****************************************************/
/*   Sample Linker Command File with Directives      */
/*****************************************************/
a.obj b.obj c.obj             /* Input filenames     */
-o prog.out -m prog.map       /* Options             */

MEMORY                        /* MEMORY directive    */
{
  RAM:  o = 100h       l = 0100h
  ROM:  o = 01000h     l = 0100h
}

SECTIONS                      /* SECTIONS directive  */
{
  .text:  {} > ROM
  .data:  {} > ROM
  .bss:   {} > RAM
}
```

Figure 9-3. An Example of a Command File with Linker Directives

The following names are reserved as key words for linker directives. Do not use them as symbol or section names in a command file.

align	l (lowercase "L")	origin
ALIGN	len	ORIGIN
block	length	page
BLOCK	LENGTH	PAGE
COPY	MEMORY	range
DSECT	NOLOAD	SECTIONS
group	o	spare
GROUP	org	

9.5 Object Libraries

An object library is a partitioned archive file that contains complete 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 TMS320C30 archiver to build and maintain archive libraries; Section 8 contains more information about the archiver.

Using object libraries can reduce linking time and can reduce the size of the executable module. If a normal object file that contains a function is specified at link time, it is linked whether it is used or not; however, if that same function is placed in an archive library, it is only included if it is referenced.

The order in which libraries are specified is important because the linker includes only those members that resolve symbols that are undefined when the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. 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 example links several object files and libraries; assume that:

- 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`.
- Library `libc.lib`, member 0, contains a definition of `origin`.
- Library `liba.lib`, member 3, contains a definition of `fillclr`.
- Member 1 of both libraries defines `clrscr`.

If you enter: `lnk30 f1.obj liba.lib f2.obj libc.lib`

then:

- Member 1 of `liba.lib` satisfies both references to `clrscr`, because the library is searched and `clrscr` is defined before `f2.obj` references it.
- Member 0 of `libc.lib` satisfies the reference to `origin`.
- Member 3 of `liba.lib` satisfies the reference to `fillclr`.

If, however, you enter: `lnk30 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. The next example creates an undefined symbol `rout1` in the linker's global symbol table:

```
lnk30 -u rout1 libc.lib
```

If any members of `libc.lib` define `rout1`, then the linker includes those members. Note that it is not possible to control the allocation of individual library members; members are allocated according to the SECTIONS directive default allocation algorithm.

Section 9.3.6 (page 9-7) describes methods for specifying directories that contain object libraries.

9.6 The MEMORY Directive

The linker determines where output sections should be allocated into memory; the linker must have a model of target memory to accomplish this task. The MEMORY directive allows you to specify a model of target memory, so you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections, and uses the model to determine which locations in the target system can be used for object code.

The memory configurations of TMS320C30 systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations to meet all applications. After you use the MEMORY directive to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

9.6.1 Default Memory Model

If you do not use the MEMORY directive, the linker uses a default memory model that is based on the TMS320C30 architecture. This model assumes that the full 24-bit address space (2^{24} locations) is present in the system and available for use.

9.6.2 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 memory range has a *name*, a *starting address*, and a *length*.

When you use the MEMORY directive, be sure to identify **all** the memory ranges that are available to load object code into. Memory that is defined by the MEMORY directive is **configured memory**; any memory that you do not explicitly account for with the MEMORY directive is **unconfigured memory**. 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 Figure 9-4 defines a system that has 4K of ROM at address 0 and 8K of RAM at address 0E000h.

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

MEMORY
{
   ROM  :   origin = 00000h   ,   length = 1000h
   RAM  :   origin = 0E000h   ,   length = 2000h
}
```

Figure 9-4. An Example of the MEMORY Directive

Now you could use the SECTIONS directive to tell the linker where to link the sections. For example, you could allocate the .text and .data sections into the area named ROM and allocate the .bss section into the area named RAM.

The general syntax of the MEMORY directive is:

MEMORY
{
 name 1 [(attr)] : **origin** = *constant* , **length** = *constant*
 name n [(attr)] : **origin** = *constant* , **length** = *constant*
}

name names a memory range. A memory name may be 1 to 8 characters; valid characters include A–Z, a–z, $, ., and —. The names have no significance to the program; they simply identify memory ranges for the linker. Memory range names are internal to the linker and are not retained in the output file or in the symbol table.

attr specifies 1 to 4 optional attributes that are associated with the named range. Valid attributes include **R** (readable memory), **W** (writable memory), **X** (executable memory), and **I** (initializable memory); attributes must be enclosed in parentheses. If you do not specify any attributes for a memory range, then the range *has all four attributes*. All memory in the default model has all four attributes. The following example defines a memory range that is readable and executable:

```
MEMORY
{   ROM (RX) : o = 0, l = 01000h   }
```

origin specifies the starting address of a memory range. It may be entered as *origin*, *org*, or *o*. The value, specified in words, is a long integer constant, and may be decimal, octal, or hexadecimal.

length specifies the length of a memory range. It may be entered as *length*, *len*, or *l*. The value is specified in words as a long integer constant (decimal, octal, or hexadecimal).

Figure 9-5 illustrates the memory map defined by Figure 9-4.

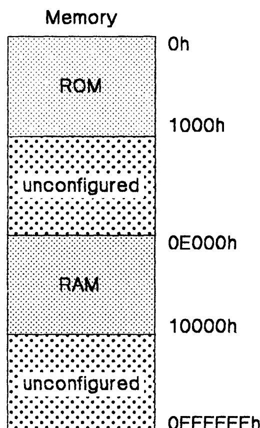


Figure 9-5. Memory Map Defined in Figure 9-4

9.7 The SECTIONS Directive

The SECTIONS directive tells the linker how to combine sections from input files into sections in the output module and where to place the output sections in memory. In summary, 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), **and**
- Permits renaming of output sections.

9.7.1 Default Sections Configuration

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. Section 9.9 (page 9-27) describes this algorithm in detail.

9.7.2 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. Figure 9-6 contains an example of the SECTIONS directive.

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

SECTIONS
{
   .text 01000h : { }

   .data : { file1.obj(.data) }

   init  :
          {
            file1.obj(init)
            file2.obj(.data)
          }

   .bss  ALIGN(16) : { }
}
```

Figure 9-6. An Example of the SECTIONS Directive

The general syntax of the SECTIONS directive is:

SECTIONS
{
   section specification 1
   section specification 2
   section specification n
}

Each section specification defines an output section. (An output section is a section in the output file.) The syntax for a section specification is:

name [binding or **align(*n*)** *]* :
    {
        input sections
        assignments
    }   *[=fill value] [> named memory]*

name — names the section in the output file. Only the first 8 characters of output section names are significant.

binding — is optional and assigns the section to a specific physical address in target memory. Section 9.7.4 (page 9-20) discusses assigning an address to an output section.

align(n) — is optional and specifies that the section should be aligned on an address boundary (the actual address is determined by the linker). Section 9.7.4 (page 9-20) discusses aligning an output section.

input sections — is a list of input sections that are combined to form the output section. The list is enclosed in braces. Section 9.7.3 (page 9-18) discusses specifying input sections in detail.

assignment — is optional and defines the value of symbols at link time or creates uninitialized spaces (called holes) between input sections within the output section. Section 9.11 (page 9-30) discusses linker assignment statements, and Section 9.12 (page 9-33) provides more information about holes.

fill value — is optional and specifies a value for filling holes in the section. See Section 9.12 (page 9-33) for more information about fill values for holes.

> named memory — Is optional and specifies that an output section should be allocated into a memory range that was named by the MEMORY directive. Section 9.7.4 (page 9-20) discusses named memory.

Figure 9-7 shows how the sections in Figure 9-6 (page 9-16) are allocated. Figure 9-6 defines four output sections, .text, .data, init, and .bss:

- The **.text** output section combines the .text sections from `file1.obj` and `file2.obj`. Notice that the braces ({ }) are empty in this section specification; this tells the linker to include all input sections that have the same name as the output section.

  An address was specified for this output section; this causes the .text output section to begin at address 01000h in the target memory (this is known as *binding*).

- The **.data** output section contains the .data section from `file1.obj`.

- The **init** section is composed of the init (named) section in `file1.obj` and the .data section in `file2.obj`.

- The **.bss** output section is composed of the .bss sections from `file1.obj` and `file2.obj`. This output section will be aligned on the next available 16-word boundary.

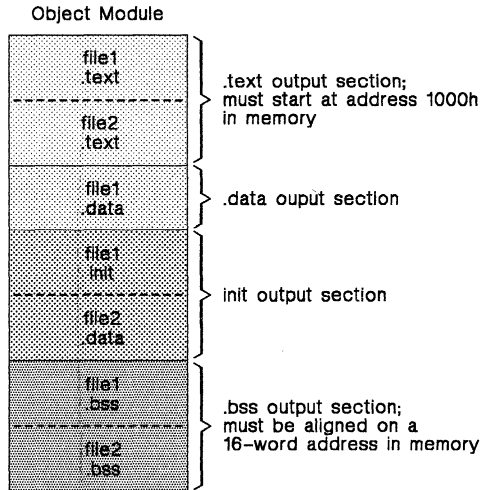


Figure 9-7. Section Allocation Defined by Figure 9-6

9.7.3 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 make up the output section.

Figure 9-8 shows the most common type of section specification; note that *no* input sections are listed.

```
SECTIONS
{
     .text  :  {   }
     .data  :  {   }
     .bss   :  {   }
}
```

Figure 9-8. The Most Common Method of Specifying Section Contents

In the example shown in Figure 9-8, the linker takes all the .text sections from the input files and combines them into the .text output section. The linker concatenates the .text input sections in the order that it encounters them in the

input files. The linker performs similar operations with the .data and .bss sections. You can use this type of specification for *any* output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
  .text :                 /* Build .text output section      */
  {
    f1.obj(.text)         /* Link .text section from f1.obj  */
    f2.obj(sec1)          /* Link sec1 section from f2.obj   */
    f3.obj                /* Link ALL sections from f3.obj   */
    f4.obj(.text, sec2)   /* Link .text and sec2 from f4.obj */
  }
}
```

Note that it is not necessary for input sections to have the same name as each other or of the output section they become part of. If a file is listed with no sections, **all** of its sections are included in the output section. If any additional input sections have the same name as an output section, but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the linker found more .text sections in the preceding example, and these .text sections *were not* specified anywhere in the SECTIONS directive, then the linker would concatenate these extra sections after `f4.obj(sec2)`.

The specifications in Figure 9-8 are actually a shorthand method for the following:

```
SECTIONS
{
      .text   :   { *(.text) }
      .data   :   { *(.data) }
      .bss    :   { *(.bss)  }
}
```

The `*(.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 certain name, but the output section name is different from the input sections' name.
- You want the linker to allocate the input sections *before* it processes additional input sections or commands within the braces.

Here's an example that uses this method:

```
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`. Note that this method includes all the *unallocated* sections. For example, if one of the .text input sections was already included in another output section when the linker encountered `*(.text)`, the linker could not include that first .text input section in the second output section.

9.7.4 Specifying the Address of an Output Section (Allocation)

After you specify the contents of each output section, you must identify the physical location in target memory where you want to load the section. Each section has an address field in its section header that tells a loader where the section should go. The process of calculating the address of the output sections is called **allocation**.

If you do not specify an explicit starting address for an output section, the linker 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 by telling the linker where a section should be loaded. You can use three methods to control section allocation:

- **Binding**

  You can supply a specific starting address for an output section by following the section name with an address:

  ```
  .text 01000h : { ... }
  ```

  This example specifies that the .text section must begin at location 1000h. The binding address must be a 24-bit constant.

  Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

  Note that 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.

- **Alignment**

  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. For example,

  ```
  SECTIONS
  {
      .data  ALIGN(32) : { ... }
  }
  ```

  In this example, the .data output section is not bound to a specific address; it is linked at the next available address in configured memory that is a multiple of 32 words.

The assembler also supports a method for specifying alignment. The .align assembler directive allows you to align code or data on a 32-word (cache) boundary. When you use .align, the assembler sets a flag that tells the linker to align the entire section. This ensures that all the alignments within the section are correct when the section is relocated.

- **Named Memory**

You can allocate a section into a memory range that was defined by the MEMORY directive. This example names ranges and links sections into them.

```
MEMORY
{
    ROM (RIX) :  origin = 0h,     length = 1000h
    RAM (RWIX):  origin = 3000h, length = 1000h
}

SECTIONS
{
    .text :                  { ... } > ROM
    .data ALIGN(64) :        { ... } > RAM
    .bss  :                  { ... } > RAM
}
```

In this example, the linker places the .text into the area called ROM. The .data and .bss output sections are allocated into RAM. You can align a section within a named memory range; the .data section is aligned on a 64-word boundary within the RAM range.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
  .text: {...} > (X)   /* .text --> executable memory    */
  .data: {...} > (RI)  /* .data --> read or init memory  */
  .bss : {...} > (RW)  /* .bss  --> read or write memory */
}
```

In this example, the .text output section can be linked into either the 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 was declared with the W attribute.

You cannot control where in the 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 other sections had been bound to addresses that would interfere with this allocation process, the .text section would start at address 0. If a section must start on a specific address, use binding instead of named memory.

9.7.5 Grouping Output Sections Together

The SECTIONS directive has a GROUP option that 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:

```
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:

When you use the GROUP option, binding, alignment, or allocation into named memory can be specified *for the group only*. You cannot use binding, named memory, or alignment for sections *within* a group.

9.8 Overlay Pages

Some target systems use an overlay memory configuration in which all or part of the memory space is overlayed by "shadow" memory. This allows the system to map different banks of physical memory in and out of a single address range in response to hardware selection signals. In this situation, multiple areas of physical memory overlay each other at one address space. You may want the linker to load various output sections into each of these areas or into areas that are not mapped at load time.

The linker supports this feature by providing *overlay pages*. Overlay pages allow you to define a memory model that has multiple address spaces. To the linker, each possible overlay configuration represents a separate address space. Each address range is treated as a separate page and must be configured separately with the MEMORY directive. You can then use the SECTIONS directive to specify which sections will be mapped into various pages.

9.8.1 Using the MEMORY Directive to Define Overlay Pages

Each separately configured address space is called a *page*. To the linker, each page represents a completely separate memory that has the full 24-bit range of addressable locations. This allows you to link two or more sections at the same (or overlapping) addresses *if they are on different pages.*

Pages are numbered sequentially, beginning with 0. Page 0 represents the "normal" address space of the TMS320C30. The default memory model resides entirely on page 0. If a memory range is specified without a page number, the linker assumes it is in page 0. This allows you to ignore the page feature for most cases; usually all sections can be linked in page 0 with no overlays.

For example, assume that your system can select between three 4K banks of physical memory to map into the address space from 1000h to 2000h. Although only one bank can be selected at a time, you can initialize each bank with different data. Assume you have three output sections called `sect0`, `sect1`, and `sect2` that must be linked into the three banks of memory. This is how you would use the MEMORY directive to obtain this configuration:

```
/*******************************************************/
/*   Example of MEMORY directive with overlay pages   */
/*******************************************************/
MEMORY
{
   PAGE 0:  ROM     : origin = 0h,       length = 1000h
            RAM     : origin = 100000h,  length = 0F00000h
            OVR_MEM : origin = 1000h,    length = 1000h
   PAGE 1:  OVR_MEM : origin = 1000h,    length = 1000h
   PAGE 2:  OVR_MEM : origin = 1000h,    length = 1000h
}
```

Figure 9-9 (page 9-24) illustrates this configuration; it shows each available block of physical memory in the system and the section that must be loaded into it.

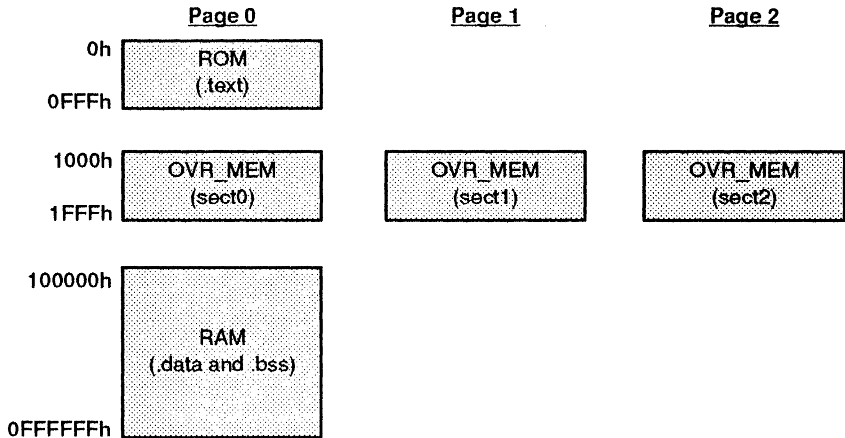


Figure 9-9. Overlay Page Example

This example defines three separate address spaces. Page 0 is the "normal" address space of the TMS320C30. It contains the memory ranges `ROM` and `RAM`; suppose they represent all the memory in the normal address space. Page 0 also contains the first bank of overlay memory (`OVR_MEM`). The other two address spaces contain only the additional banks of overlay memory, both labeled `OVR_MEM`. Note that all three `OVR_MEM` ranges cover the same address range. This is possible because each range is on a different page and therefore represents a different memory space.

9.8.2 Using Overlay Pages with the SECTIONS Directive

The SECTIONS directive allows you to tell the linker which page an output section should be linked into. Each output section of the program is assigned a page as well as an address. You can assign an output section to an overlay page by following the section specification with the PAGE option and a page number. Continuing the example from the previous discussion, the SECTIONS definition would be:

```
SECTIONS
{
   .text: {} > ROM              /* Link .text in ROM on page 0     */
   .data: {} > RAM              /* Link .data in RAM on page 0     */
   .bss : {} > RAM              /* Link .bss in RAM on page 0      */
   sect0: {} > OVR_MEM PAGE 0   /* Link sect0 into bank 0 (page 0) */
   sect1: {} > OVR_MEM PAGE 1   /* Link sect1 into bank 1          */
   sect2: {} > OVR_MEM PAGE 2   /* Link sect2 into bank 2          */
}
```

If you don't specify a page number for an output section, the linker assumes page 0. In this example, .text, .data, and .bss are all linked into the named memory areas on page 0. (The PAGE 0 could have been omitted from the `sect0` definition as well.)

The PAGE specifications for `sect0`, `sect1`, and `sect2` tell the linker to link these output sections into the corresponding overlay pages. As a result, they all are linked to address 1000h, *but in different memory spaces.* When the

program is loaded, a loader can configure hardware in such a way that each of these sections is loaded into the appropriate bank of memory.

Within a page, you can bind output sections or use named memory areas in the usual way. In the preceding example, notice how `sect1` is allocated into the memory range `OVR_MEM`. This allows you to define the allocation of sections within a page, just as you can in a single memory space.

For example, the following statement:

```
sect1 1200h: {} PAGE 1
```

links `sect1` at address 1200h in page 1. If you do not specify any binding or named memory range for the section, the linker allocates the section into the page wherever it can (just as it normally does with a single memory space). For example, `sect2` could also be specified as:

```
sect2 : {} PAGE 2
```

Because `OVR_MEM` is the only memory on page 2, it is not necessary (but acceptable) to specify `>OVR_MEM` for the section.

9.8.3 Page Definition Syntax

As illustrated in the preceding examples, overlay pages are specified in the MEMORY directive by using the following syntax:

MEMORY
{
   PAGE 0 :   *memory range*
                memory range

   PAGE *n* :   *memory range*
                memory range
}

Each page is introduced by the keyword PAGE and a page number, followed by a colon and a list of memory ranges the page contains. Memory ranges are specified in the normal way. You can define up to 255 overlay pages. Because each page represents a completely independent address space, memory ranges on different pages can have the same name. Configured memory on any page can overlap configured memory on any other page. *Within a single page, however, all memory ranges must have unique names and must not overlap.*

Any memory ranges listed outside the scope of a PAGE specification default to page 0. Consider the following example:

```
MEMORY
{
             ROM    : org =     0h   len =  1000h
             EPROM  : org =  1000h   len =  1000h
             RAM    : org =  2000h   len = 0E000h
   PAGE 1:   XROM   : org =     0h   len =  1000h
             XRAM   : org =  2000h   len = 0E000h
}
```

The memory ranges `ROM`, `EPROM`, and `RAM` are all on page 0 (because no page is specified). `XROM` and `XRAM` are on page 1. Note that `XROM` on page 1 overlays `ROM` on page 0 and `XRAM` on page 1 overlays `RAM` on page 0.

In the output link map (obtained with the -m linker option), the listing of the memory model is keyed by pages. This provides you with an easy method of verifying that you specified the memory model correctly. Also, the listing of output sections has a PAGE column that identifies the memory space into which each section will be loaded.

9.9 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 has default algorithms that it uses to build and allocate sections, within the specifications you supply. Section 9.9.1 and Section 9.9.2 describe default allocation algorithms.

9.9.1 Allocation Algorithm

If you do not use the MEMORY directive, the linker assumes that the full 24-bit address space is configured and allocates output sections into memory beginning at address 0.

If you do not use the SECTIONS directive, the linker allocates the output sections as though the following SECTIONS directive was specified:

```
SECTIONS
{
        .text : { }
        .data : { }
        .bss  : { }
}
```

All .text input sections are concatenated to form a .text output section in the executable output file. All .data input sections are combined to form a .data output section, and all .bss sections are combined to form a .bss output section. Each output section is then allocated into configured memory.

If the input files contain named sections the linker links them in after the .bss section. Input sections that have the same name are combined into a single output section with this name.

> **Note:**
>
> When you use the 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 rules discussed in Section 9.9.2.

9.9.2 General Rules for Output Sections

An output section can be formed in one of two ways:

Rule 1: As the result of a SECTIONS directive definition.

Rule 2: By combining input sections with the same names into output sections that are not defined in a SECTIONS directive.

If an output section is formed as a result of a SECTIONS directive (rule 1), this definition completely determines its contents. (See Section 9.7, page 9-16, for examples of how to specify the contents of output sections.)

An output section can also be formed when input sections are encountered that are not specified by any SECTIONS directive (rule 2). In this case, the

linker combines all such input sections that have the same name into an output section with this 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 to contain them. The linker will combine the two `Vectors` sections from the input files into a single output section named `Vectors`, allocate it into memory, and include it in the output file.

After the linker determines the composition of all the output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured, or if there is no MEMORY directive, the linker uses the default configuration.

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. This is the algorithm:

1) Any output section for which you have listed a specific binding address is placed in memory at that address.

2) Any 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 were defined. Sections not defined in a SECTIONS directive are allocated in the order in which they were encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

9.10 Special Section Types (DSECT, COPY, and NOLOAD)

You can assign three special types to output sections: DSECT, COPY, and NOLOAD. These types affect the way that the program is treated when it is linked and loaded. You can assign a type to a section by placing the type (enclosed in parentheses) after the section definition. For example,

```
SECTIONS
{
        sec1 200000h (DSECT)  : {f1.obj}
        sec2 400000h (COPY)   : {f2.obj}
        sec3 600000h (NOLOAD) : {f3.obj}
}
```

- The DSECT type creates a "dummy section" with the following qualities:
  - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.
  - It can overlay other output sections, other DSECTs, and unconfigured memory.
  - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.
  - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.
  - The section's contents, relocation information, and line number information are not placed in the output module.

  In the preceding example, none of the sections from `f1.obj` are allocated, but all the symbols are relocated as though the sections were linked at address 200000h. 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 C compiler has this attribute under the RAM model.

- A NOLOAD section differs from a normal output section in one respect: the section's contents relocation information and line number information are not placed in the output module. The linker allocates space for the section, the section is listed in the memory map listing, etc.

9.11 Assigning Symbols at Link Time

Linker assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to assign an allocation-dependent value to a variable or a pointer.

9.11.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of C assignment statements:

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 in the program. If it is not, the linker defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in Section 9.11.3. Assignment statements **must** be terminated with a semicolon.

The linker processes assignment statements **after** it allocates all the output sections. Thus, 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; `cur_tab` must point to `Table1` or `Table2`. You could accomplish this in the assembly code, but you would need to reassemble the program in order to change tables. Instead, you can use a linker assignment statement to assign `cur_tab` at link time:

```
prog.obj             /* Input file                          */
cur_tab = Table1;    /* Assign cur_tab to one of the tables */
```

9.11.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the SPC during allocation. The linker's "." symbol is analogous to the assembler's "$" symbol. The "." symbol can only be used in assignment statements within a SECTIONS directive, because "." is only meaningful during allocation, and the allocation process is controlled by the SECTIONS directive.

For example, suppose a program needs to know the address of the beginning of the .data section. You can create an external undefined variable `Dstart` in the program by using the .global directive. Then, assign the value of "." to `Dstart`:

```
SECTIONS
{
   .text: {}
   .data: { Dstart = .; }    /* Dstart = current SPC value */
   .bss : {}
}
```

This defines `Dstart` to be the ultimate linked address of the .data section. The linker will relocate all references to `Dstart`.

A special type of assignment assigns a value to the "." symbol. This adjusts the location counter 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 ".". Assignments to "." and holes are described in Section 9.12.

9.11.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in Table 9-2.
- All numbers are treated as long (32-bit) integers.
- Constants are identified in the same manner as they are by the assembler. That is, numbers are recognized as decimals 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, the symbol is relocatable; if assigned the value of an absolute expression, the symbol is absolute.

The linker supports the C language operators listed in Table 9-2 in order of precedence. Operators in the same group have the same precedence.

Besides the operators listed in Table 9-2, the linker also has an *align* operator that allows a symbol to be aligned on an *n*-word 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-word boundary. Because the align operator is a function of the current SPC, it can only be used in the same context as "." - that is, within a SECTIONS directive.

Table 9-2. Operators in Assignment Expressions

Group 1 (Highest Precedence)		Group 6	
!	Logical Not	&	Bitwise AND
~	Bitwise Not		
-	Negative		
Group 2		**Group 7**	
*	Multiplication	|	Bitwise OR
/	Division		
%	Mod		
Group 3		**Group 8**	
+	Addition	&&	Logical AND
-	Minus		
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		

9.11.4 Symbols Defined by the Linker

The linker automatically defines three symbols that a program can use at run time to determine where a section is linked. These symbols are external, so they appear in the link map. They can be accessed in any assembly language module if they are declared with a .global directive.

Values are assigned to these symbols as follows:

.text is assigned the first address following 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 following 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.)

cinit is assigned the first address of the .cinit section (when -c or -cr is used).

9.12 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called **holes**. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles such holes and how you can fill holes (and uninitialized sections) with a value.

9.12.1 Initialized and Uninitialized Sections

There are two rules to remember about the contents of an output section. An output section contains:

Rule 1: Raw data for the *entire* section **or**

Rule 2: *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 or .asect assembler directives also have raw data.

By default, the .bss section and .usect sections 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 RAM for variables. In the object file, an uninitialized section has a normal section header and may have symbols defined in it; however, no memory image is stored in the section.

9.12.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections when building an output section. When such a hole is created, *the linker must follow rule 1 and supply raw data for the hole.*

Holes can only be created *within* output sections. There can also be space *between* output sections, but such spaces are not holes. There is no way to fill or initialize space between output sections.

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 either adding to it, assigning a greater value to it, or aligning it to an address boundary. The operators, expressions, and syntax of assignment statements are described in Section 9.11 (page 9-30).

The following example shows how holes can be created in output sections using assignment statements:

```
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
   }
}
```

The output section outsect is built as follows:

- The .text section from file1.obj is linked in.
- The linker creates a 256-word hole.
- The .text section from file2.obj is linked in after the hole.
- The linker creates another hole by aligning the SPC on a 16-word boundary.
- 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 specify a binding address). Consider the statement . = align(16) in the preceding example. This statement effectively aligns file3.obj .text to start on a 16-word boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, then file3 .text will not be aligned either.

Expressions that decrement "." are illegal. For example, it is invalid to use the -= operator in an assignment to ".". The most common operators used in assignments to "." 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 initialized sections to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* An example of creating a hole in this way is:

```
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 (rule 1). Therefore, the uninitialized .bss section becomes a hole.

Note that uninitialized sections only become holes when they are combined with initialized sections. If multiple uninitialized sections are linked together the resulting output section is also uninitialized.

9.12.3 Filling Holes

Whenever there is a hole in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 4-byte 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 that specific initialized section. Follow the section name with an = symbol and a 4-byte constant:

```
SECTIONS
{
      outsect:
      {
           file1.obj(.text)
           file2.obj(.bss) = 0FFh   /* Fill this hole */
      }                             /* with 000000FFh */
}
```

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. For example,

```
SECTIONS
{
     outsect:
     {
          . += 10h;              /* This creates a hole          */
          file1.obj(.text)
          file1.obj(.bss)        /* This creates another hole   */
     } = 0FF00h                  /* This fills both holes with  */
}                                /* 0000FF00h                   */
```

3) If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with -f. 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:

```
lnk30 -f 0FFFFFFFFh link.cmd
```

This fills the hole with 0FFFFFFFFh.

4) If you do not invoke the linker with -f, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

9.12.4 Explicit Initialization of Uninitialized Sections

An uninitialized section only becomes a hole when it is combined with an initialized section. When uninitialized sections are combined with each other, the resulting output section remains uninitialized and has no raw data in the output file.

However, you can force an uninitialized section to be initialized simply 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: {} = 11223344h  /* Fills .bss with 11223344h */
}
```

> **Note:**
>
> Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large sections or holes.

9.13 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:

- Intermediate files **must** have relocation information. Use the -r option when you link the file the first time.
- 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.
- Intermediate link steps should only be concerned with the formation of output sections, and not with allocation. All allocation, binding, and MEMORY directives should be performed in the final link.

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`.

  ```
  lnk30 -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`.

  ```
  lnk30 -r -o tempout2 file2.com
  ```

  `file2.com` contains:

  ```
  SECTIONS
  {
     ss2:{
           g1.obj
           g2.obj
             .
             .
             .
           gn.obj
         }
  }
  ```

- **Step 3:** Link `tempout1.out` and `tempout2.out`:

  ```
  lnk30 -m final.map -o final.out tempout1.out tempout2.out
  ```

9.14 Linking C Code

The TMS320C30 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`:

```
lnk30 -c -o prog.out prog1.obj prog2.obj ... rts.lib
```

The -c option tells the linker to use special conventions that are defined by the C environment. The archive library `rts.lib` contains C runtime support functions.

For more information about C, including the runtime environment and runtime support functions, see the *TMS320C30 C Compiler Reference Guide*.

9.14.1 Runtime Initialization

All C programs must be linked with an object module called `boot.obj`, which contains code and data for initializing the runtime environment.

When the program begins running, this code is executed first and performs the following actions:

- Sets up the system stack
- Processes the runtime initialization table and autoinitializes global variables (in the ROM model)
- Disables interrupts and calls `_main`

The runtime support object library, `rts.lib`, contains `boot.obj`. You can use the archiver to extract `boot.obj` from the library, and then link it in directly, or you can simply include `rts.lib` as an input file and the linker will extract `boot.obj` when you use the -c or -cr option.

9.14.2 Object Libraries and Runtime Support

The *TMS320C30 C Compiler Reference Guide* describes additional runtime support functions that are included in `rts.lib`. If your program uses any of these functions, you must link `rts.lib` with your object files.

You can also create your own object libraries and link them. The linker will include and link only those modules in a library that resolve undefined references.

9.14.3 Autoinitialization (ROM and RAM Models)

The C compiler produces tables of data that are used to autoinitialize global variables. These are contained in a special section called **.cinit**. The initialization tables can be used for autoinitialization in either of two ways.

- **ROM Model** (-c option)

  Variables are initialized at *run time*. The .cinit section is loaded into memory along with all the other sections. The linker defines a special symbol called `cinit` that points to the beginning of the tables in mem-

ory. When the program begins running, the C boot routine copies data from the tables into the specified variables in the .bss section. This allows initialization data to be stored in ROM and then copied to RAM each time the program is started.

Figure 9-10 illustrates the ROM autoinitialization model.

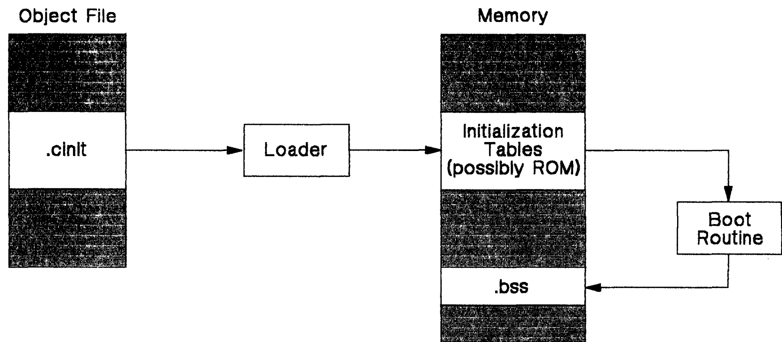


Figure 9-10. ROM Model of Autoinitialization

- **RAM Model** (-cr option)

  Variables are initialized at *load time*. This can enhance performance by reducing boot time and can save memory used by the initialization tables. (Note that you must use a smart loader to take advantage of the RAM model of autoinitialization.)

  When you use -cr, the linker marks the .cinit section with a special attribute. This attribute tells the linker *not* to load the .cinit section into memory. The linker also sets the `cinit` symbol to -1; this informs the C boot routine that initialization tables *are not* present in memory. Thus, no runtime initialization is performed at boot time.

  When the program is loaded, the loader must be able to:

  - Detect the presence of the .cinit section in the object file.
  - Detect the presence of the attribute that tells it not to copy the .cinit section.
  - Understand the format of the initialization tables (this is described in the *TMS320C30 C Compiler Reference Guide*).

  The loader then uses the initialization tables directly from the object file to initialize variables in .bss.

  Figure 9-11 (page 9-40) illustrates the RAM autoinitialization model.

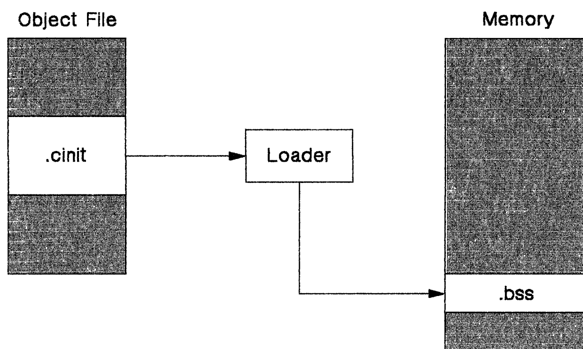


Figure 9-11. RAM Model of Autoinitialization

9.14.4 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. `_c_int00` is the start of the C boot routine in `boot.obj`; referencing `_c_int00` ensures that `boot.obj` will automatically be linked in from the runtime support library `rts.lib`.
- The .cinit output section is padded with a termination record so that the boot routine (ROM model) or the loader (RAM model) knows when to stop reading the initialization tables.
- In the ROM model (-c option), the linker defines the symbol `cinit` as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.
- In the RAM model (-cr option):
  - The linker sets the symbol `cinit` to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at boot time.
  - The STYP_COPY flag (010h) 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.

9.15 Linker Example

This example links three object files named `demo.obj`, `fft.obj` and `tables.obj` and creates a program called `demo.out`. The symbol `SETUP` is the program entry point.

Assume that target memory has the following configuration:

Address Range	Contents
000000h to 000FFFh	4K on-chip ROM
801000h to 8013FFh	Internal RAM block B0
801400h to 8017FFh	Internal RAM block B1
801800h to 0FFFFFFh	External RAM

The output sections are constructed from the following input sections:

- A set of interrupt vectors from section `int_vecs` in the file `tables.obj` must be linked at address 0 in ROM.
- Executable code, contained in the .text sections of `demo.obj` and `fft.obj`, must also be linked into ROM.
- Two tables of coefficients, which are in the .data sections of the files `tables.obj` and `fft.obj` must be linked into RAM block B0. The remainder of block B0 must be initialized to the value 0FFCC1122h.
- The .bss section from `fft.obj`, which contains variables, must be linked into block B1 of data RAM. The unused part of this RAM must be initialized to 0FFFFFFFFh.
- The .bss section from `demo.obj`, which contains buffers and variables, must be linked into external RAM.

Figure 9-12 shows the linker command file for this example; Figure 9-13 shows the map file.

```
/***********************************************************************/
/****                  Specify Linker Options                      ****/
/***********************************************************************/

-e SETUP                        /*   Define the entry point            */
-o demo.out                     /*   Name the output file              */
-m demo.map                     /*   Create a load map                 */

/***********************************************************************/
/****                 Specify the Input Files                      ****/
/***********************************************************************/

demo.obj
fft.obj
tables.obj

/***********************************************************************/
/****             Specify the Memory Configuration                 ****/
/***********************************************************************/

MEMORY
{
    ROM:       origin = 0000000h   length = 01000h
    RAM_B0:    origin = 0801000h   length = 0400h
    RAM_B1:    origin = 0801400h   length = 0400h
    RAM        origin = 0801800h   length = 07FE800h
}
/***********************************************************************/
/****               Specify the Output Sections                    ****/
/***********************************************************************/


SECTIONS
{
    .text: {}   >ROM            /* Link all .text sections into ROM    */

    int_vecs 0h: {}             /* Link interrupts at 0                */

    .data:                      /* Link the .data sections             */
    {
        tables.obj(.data)       /* .data input section                 */
        fft.obj(.data)          /* .data input section                 */
        . = 400h;               /* Create a hole to end of block       */
    } = 0FFCC1122h > RAM_B0     /* Fill and link into B0               */

    fftvars:                    /* Create a new fftvars section        */
    {
        fft.obj(.bss)
    } = 0FFFFFFFFh > RAM_B1     /* Fill and link into B1               */

    .bss: {}    >RAM            /* Link all remaining .bss sections    */
}

/***********************************************************************/
/****                   End of Command File                        ****/
/***********************************************************************/
```

Figure 9-12. Linker Command File, demo.cmd

Invoke the linker with the following command:

`lnk30 demo.cmd`

This creates the map file shown in Figure 9-13 and an output file called `demo.out` that can be run on the TMS320C30.

```
*****************************************************************
TMS320C30 COFF Linker, Version 1.00, 87.070
*****************************************************************

OUTPUT FILE NAME:    <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address:  00000040


MEMORY CONFIGURATION
           name      origin           length          attributes
         --------   --------        ---------         ----------
         ROM        00000000        000001000            RWIX
         RAM_BO     00801000        000000400            RWIX
         RAM_B1     00801400        000000400            RWIX
         RAM        00801800        0007FE800            RWIX


SECTION ALLOCATION MAP

 output                                         attributes/
section    page   origin        length            input sections
--------   ----  ----------   -----------     ----------------
int_vecs    0    00000000     00000040
                   00000000     00000040         demo.obj (int_vecs)

.text       0    00000040     000001B0
                   00000040     0000014E       demo.obj (.text)
                   0000018E     00000064         fft.obj (.text)

.data       0    00801000     00000400
                   008010000    000000A5         tables.obj (.data)
                   0080100A5    00000014         fft.obj (.data)
                   0080100B9    00000347         --HOLE-- [fill = ffcc122]

fftvars     0    00801400     0000001A
                   00801400     0000001A         fft.obj (.bss) [fill = ffffffff]

.bss        0    00801800     0000009A           UNINITIALIZED
                   00801800     0000009A         demo.obj (.bss)


GLOBAL SYMBOLS
 address    name                                      address    name
---------   ----                                     ---------   ----
00000040    SETUP                                    00000040    SETUP
00801400    edata                                    0000004A    start
0080180A    end                                      0000008A    fft
000001F2    etext                                    00000120    sub
00801800    extvar                                   00000166    list
0000008A    fft                                      00000170    p1asm
00000166    list                                     0000017A    p2asm
00000184    main                                     00000184    main
00000170    p1asm                                    000001F2    etext
0000017A    p2asm                                    00801400    edata
0000004A    start                                    00801800    extvar
00000120    sub                                      0080189A    end

[12 symbols]
```

Figure 9-13. Output Map File, demo.map

Section 10

Object Format Converter Description

Most EPROM programmers do not accept COFF object files as input. The object format converter converts a COFF object file into one of three object formats that most EPROM programmers accept as input:

- Tektronix hex object format
- Intel hex object format
- TI-tagged object format

The object format converter accepts one COFF object file as input. If you are converting to TI-tagged object format, the utility produces one output file. If you are converting to Tektronix or Intel object format, the utility produces four output files (one output file for each set of bytes, from least significant to most significant bytes).

This section contains the following topics:

Section **Page**

10.1 Object Format Converter Development Flow ... 10-2
10.2 Invoking the Object Format Converter ... 10-3
10.3 Examples ... 10-4
10.4 Halt Conditions ... 10-4

10.1 Object Format Converter Development Flow

Figure 10-1 illustrates the object format converter's role in the assembly language development process.

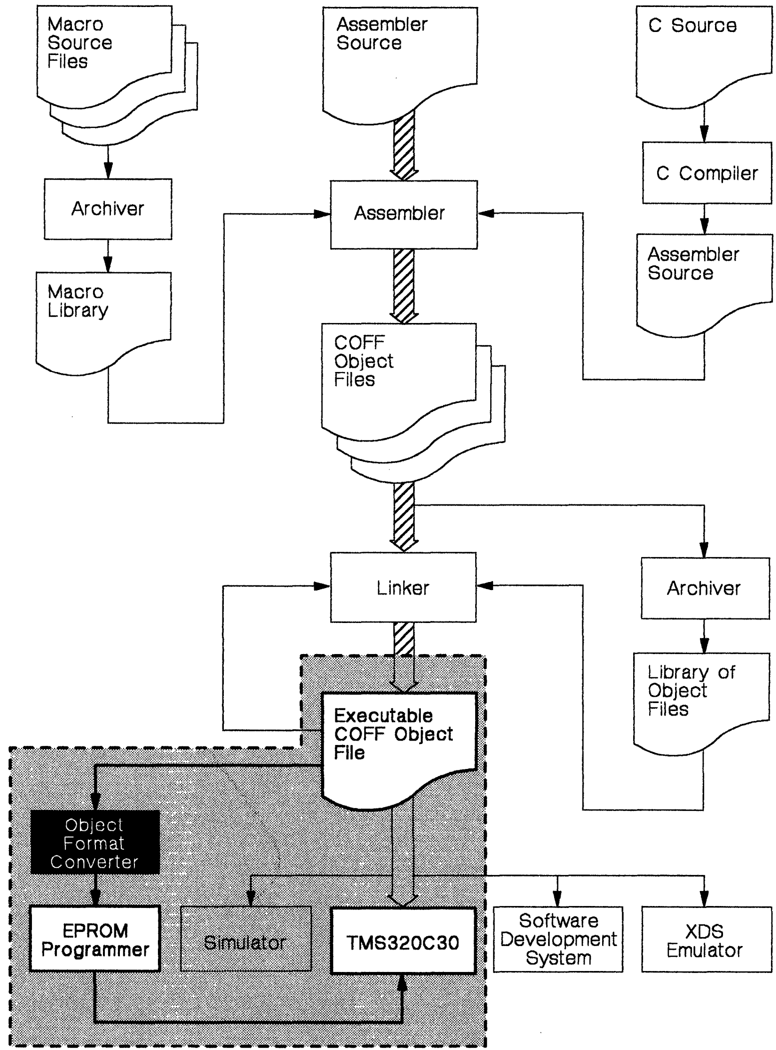


Figure 10-1. Object Format Converter Development Flow

10.2 Invoking the Object Format Converter

To invoke the object format converter, enter:

rom30 *[-option] [COFF input file]*

rom30 is the command that invokes the object format converter; all parameters are optional.

The *filename* is the name of the file that you want to convert. If you do not specify an input filename, the object format converter prompts for one. If you specify a filename without an extension, the utility assumes that the filename has a default extension of ***.obj***.

There are three *options* which can be entered anywhere on the line. The options identify the format of the output file:

-i specifies Intel hex object format for the output.

-t specifies TI-tagged object format for the output.

-x specifies Tektronix hex object format for the output.

If you don't supply an option, the object format converter produces Tektronix hex format output files. The object format converter uses the input filename (without it's extension) to name output files; it chooses the file extension depending on the type of output you've requested:

- For *TI-tagged* format, the object format converter produces one output file with an extension of ***.tag***.
- For *Intel* or *Tektronix* format, the object format converter produces four files with the following extensions:
  - ***.b0*** is the extension for the file that contains the least significant bytes.
  - ***.b1*** is the extension for the file that contains the next-to-least significant bytes.
  - ***.b2*** is the extension for the file that contains the next-to-most significant bytes.
  - ***.b3*** is the extension for the file that contains the most significant bytes.

When the object format converter finishes converting the input file, it prints the message `Translation complete.`

10.3 Examples

Here are some examples of using the object format converter.

- **Example 1:**

  You can invoke the object format converter with no options and no filename:

  ```
  rom30
  ```

  The utility will print the following banner and prompt:

  ```
  COFF Object Converter    Version 5.01, 87.610
  (c) Copyright 1987, Texas Instruments Inc.

      Coff file [.obj]:
  ```

  If, for example, you respond to the prompt with a filename of `fft`, the object format converter uses the file `fft.obj` as an input file. It produces four Tektronix-format output files named `fft.b0`, `fft.b1`, `fft.b2`, and `fft.b3`.

- **Example 2:**

  If you enter:

  ```
  rom30 -i in
  ```

  the utility uses `in.obj` as the input file. It creates four Intel-format files named `in.b0`, `in.b1`, `in.b2`, and `in.b3`.

- **Example 3:**

  If you enter:

  ```
  rom30 in.tmp -t
  ```

  the object format converter uses `in.tmp` as the input file. It produces a single TI-tagged output file named `in.tag`.

10.4 Halt Conditions

There are two situations in which the object format converter aborts execution:

1) If any of the specified files cannot be opened, the code conversion utility prints the message `Input COFF file cannot be opened` and aborts.

2) If you supply the utility with the name of an invalid object file, the object format converter prints the message `Corrupt input file` and aborts.

Appendix A

Common Object File Format

The TMS320C30 assembler and linker create object files that are 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 object file format has been chosen because it encourages modular programming and provides more powerful and flexible methods for managing code segments and target system memory.

One of the basic COFF concepts is *sections*. Section 3, Introduction to Common Object File Format, discusses COFF sections in detail. If you understand section operation, you will be able to use the TMS320C30 assembly language tools more efficiently.

This appendix contains technical details about COFF object file structure. Most of this information pertains to the symbolic debugging information that is produced by the TMS320C30 C compiler. The main purpose of this appendix is to provide supplementary information for those of you who are interested in the internal format of object files.

Topics in this appendix include:

Section **Page**

A.1 File Structure A-2
A.2 File Header A-4
A.3 Optional File Header A-5
A.4 Section Headers A-6
A.5 Relocation Information A-8
A.6 Line Number Table A-9
A.7 Symbol Table A-11

A.1 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 section (except .bss),
- Relocation information for each section (except .bss),
- Line number entries for each section (except .bss),
- A symbol table, **and**
- 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 will not contain relocation entries. Figure A-1 illustrates the overall object file structure.

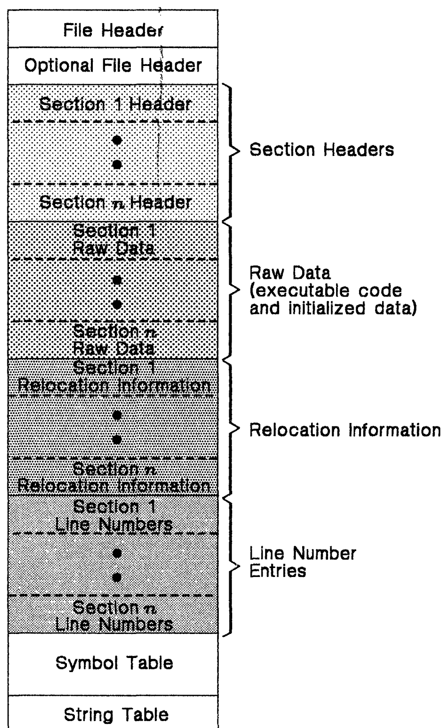


Figure A-1. COFF File Structure

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 .text, .data, and .bss sections, respectively, are placed in the object file, followed by any named sections in the order in which they were encountered. Although the .bss section has a section header, notice that it has no raw data, no relocation information, and no line number entries; named sections created with .usect do not have this type of information, either. This is because the .bss and .usect directives simply reserve space for uninitialized data; their sections contain no actual code.

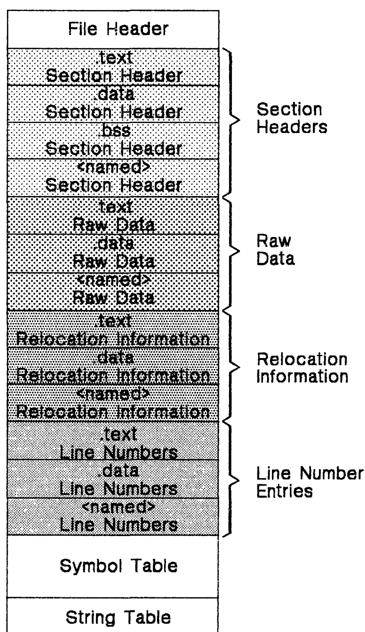


Figure A-2. Sample COFF Object File

A.2 File Header

The file header contains 20 bytes of information that describe the general format of an object file. Table A-1 shows the structure of the file header.

Table A-1. File Header Contents

Byte Number	Type	Description
0–1	Unsigned short integer	Magic number (093h), indicates that the file can be executed in a TMS320C30 system
2–3	Unsigned short integer	Number of section headers
4–7	Long integer	Time and date stamp, indicates when the file was created
8–11	Long integer	File pointer, contains the symbol table's starting address
12–15	Long integer	Number of entries in the symbol table
16–17	Unsigned short integer	Number of bytes in the optional header. This field is either 0 or 28; if it is 0, then there is no optional file header.
18–19	Unsigned short integer	Flags (see Table A-2)

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, then F_RELFLG and F_EXEC are both set.)

Table A-2. File Header Flags (Bytes 18 and 19)

Mnemonic	Flag	Description
F_RELFLG	0001h	Relocation information was stripped from the file
F_EXEC	0002h	The file is executable (it contains no unresolved external references)
F_LNNO	0004h	Line numbers were stripped from the file
F_LSYMS	0010h	Local symbols were stripped from the file
F_AR32WR	0040h	The file has the byte ordering used by the TMS320C30 support tools (32 bits per word, least significant byte first)

A.3 Optional File Header

The linker creates the optional file header and uses it to perform relocation at download 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 integer	Magic number (0108h)
2–3	Short integer	Version stamp
4–7	Long integer	Size (in words) of executable code
8–11	Long integer	Size (in words) of initialized words
12–15	Long integer	Size (in bits) of uninitialized data
16–19	Long integer	Beginning address of executable code
24–27	Long integer	Beginning address of initialized data

A.4 Section Headers

COFF object files contain a table of section headers that specify where each section begins in the object file. Each section of the file has its own section header.

Table A-4. Section Header Contents

Byte Number	Type	Description
0–7	Character	Eight-character section name, padded with nulls
8–11	Long integer	Section's physical address
12–15	Long integer	Section's virtual address
16–19	Long integer	Section size in words
20–23	Long integer	File pointer to raw data
24–27	Long integer	File pointer to relocation entries
28–31	Long integer	File pointer to line number entries
32–33	Unsigned short integer	Number of relocation entries
34–35	Unsigned short integer	Number of line number entries
36–37	Unsigned short integer	Flags (see Table A-5)
38	Character	Reserved.
39	Character	Memory page number.

Table A-5 lists the flags that can appear in bytes 36 and 37 of the section header.

Table A-5. Section Header Flags (Bytes 36 and 37)

Mnemonic	Flag	Description
STYP_REG	0000h	Regular section (allocated, relocated,loaded)
STYP_DSECT	0001h	Dummy section (relocated, not allocated, not loaded)
STYP_NOLOAD	0002h	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0004h	Grouped section (formed from several input sections)
STYP_PAD	0008h	Padding section (loaded, not allocated, not relocated)
STYP_COPY	0010h	Copy section (not allocated, relocated, loaded; relocation and line number entries are processed normally)
STYP_TEXT	0020h	Section contains executable code
STYP_DATA	0040h	Section contains initialized data
STYP_BSS	0080h	Section contains uninitialized data
STYP_ALIGN	0100h	Section is aligned on a cache boundary

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 flags word is set to 024h, then both STYP_GROUP and STYP_TEXT are set.

Figure A-6 illustrates how the pointers in a section header would point to the various elements in an object file that are associated with the .text section.

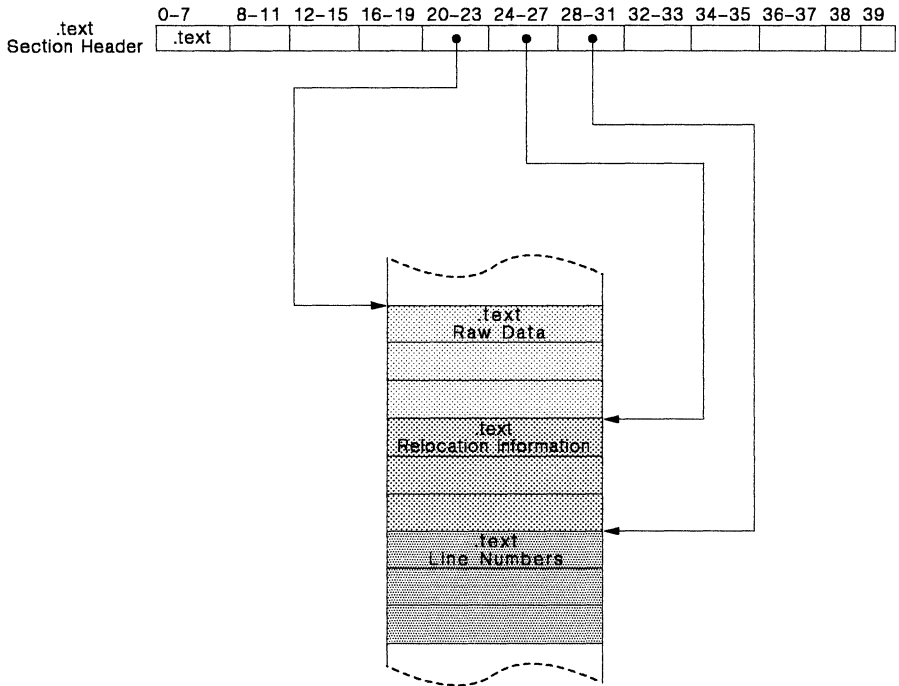


Figure A-3. An Example of Section Header Pointers for the .text Section

As Figure A-2 (page A-3) shows, the .bss section and uninitialized sections defined with .usect vary from this format. Although there is a section header for each uninitialized section, these sections have no raw data, no relocation information, no line number information, and occupy no actual space in the object file. Therefore, the number of relocation entries, the number of line number entries, and the file pointers in an uninitialized section header are zero. Section headers for uninitialized sections simply tell the linker how much space for variables it should reserve in the memory map.

A.5 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 an input section are treated.

The relocation information entries use the 10-byte format shown in Table A-6.

Table A-6. Relocation Entry Contents

Byte Number	Type	Description
0–3	Long integer	Virtual address of the reference
4–5	Unsigned short integer	Symbol table index (0–65535)
6–7	Unsigned short integer	Reserved
8–9	Unsigned short integer	Relocation type (see Table A-7)

Table A-7 lists the relocation types that can appear in bytes 8 and 9 of the relocation entry.

Table A-7. Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_REL24	005h	24-bit direct reference
R_RELWORD	0010h	16-bit direct reference to symbol's address
R_RELLONG	0011h	32-bit direct reference to symbol's address
R_PCRWORD	0013h	16 bits, PC relative
R_OCRLONG	0018h	1's complement 32-bit direct
R_GSPPCR16	0019h	16-bit relative (in words)
R_PARTLS16	0020h	Truncate to lower 16 bits
R_PARTMS8	0021h	Relocate bits 24 through 16

A.6 Line Number Table

The object file contains a table of line number entries that are useful for symbolic debugging. When the C compiler produces several lines of assembly language code, it creates a line number entry that maps these lines back to the original line of C source code that generated them. Each single line number entry contains 6 bytes of information. Table A-8 shows the format of a line number entry.

Table A-8. Line Number Entry Format

Byte Number	Type	Description
0–3	Long integer	This entry may have one of two values: 1) If it is the first entry in a block of line number entries, it points to a symbol entry in the symbol table 2) If it is not the first entry in a block, it is the physical address of the line indicated by bytes 4–5
4–5	Unsigned short integer	This entry may have one of two values: 1) If this field is 0, then this is the first line of a function entry 2) If this field is *not* 0, then this is the line number of a line of C source code

Figure A-4 shows how line number entries are grouped into blocks.

Symbol Index	**0**
physical address	line number
physical address	line number
. . .	. . .
Symbol Index	**0**
physical address	line number
physical address	line number

Figure A-4. Line Number Blocks

As Figure A-4 shows, each entry is divided into halves:

- For the *first line* of a function,
  - Bytes 0–3 point to the name of a symbol or a function in the symbol table.
  - Bytes 4–5 contain a 0, which indicates the beginning of a block.
- For the *remaining lines* in a function,
  - Bytes 0–3 show the physical address (the number of words created by a line of C source).
  - Bytes 4–5 show the address of the original C source, relative to its appearance in the C source program.

The line entry table can contain many of these blocks.

Figure A-9 illustrates an example of line number entries for a function named XYZ. As shown, the function name is entered as a symbol in the symbol table. The first portion on XYZ's block of line number entries points to the function name in the symbol table. Assume that the original function in the C source contained three lines of code. The first line of code produces 4 words of assembly language code, the second line produces 3 words, and the third line produces 10 words. Figure A-9 shows what the line number entries would look like for this example.

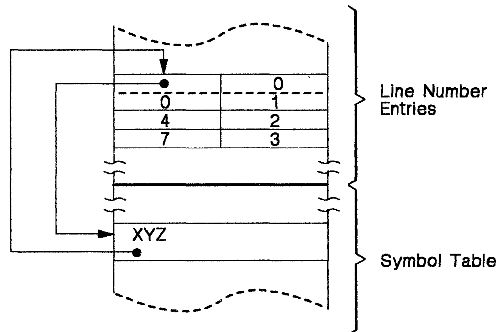


Figure A-5. Line Number Entries Example

(Note that the symbol table entry for XYZ has a field that points back to the beginning of the line number block.)

Since line numbers are not often needed, the linker provides an option (-s) that strips line number information from the object file. This provides a more compact object module.

A.7 Symbol Table

The order of symbols in the symbol table is very important; they appear in the sequence shown in Figure A-3.

File Name 1
Function 1
Local symbols for Function 1
Function 2
Local symbols for Function 2
. . .
File Name 2
Function 1
Local symbols for Function 1
. . .
Static variables
. . .
Defined global symbols
Undefined global symbols

Figure A-6. Symbol Table Contents

Static variables refer to symbols defined in C that have storage class static outside any function. If you have several modules that use symbols with the same name, making them static confines the scope of each symbol to the module that defines it (this eliminates multiple-definition conflicts).

The entry for each symbol in the symbol table contains the symbol's:

- Name (or a pointer into the string table)
- Type
- Value
- Section it was defined in
- Storage class
- Basic type (integer, character, etc.)
- Derived type (array, structure, etc.)
- Dimensions
- Line number of the source code that defined the symbol

Section names are also defined in the symbol table.

All symbol entries, regardless of the symbol's class and type, have the same format in the symbol table. Each symbol table entry contains the 18 bytes of information listed in Table A-9. Some symbols may not have all the characteristics listed above; if a particular field is not set, it will be set to null.

Table A-9. Symbol Table Entry Contents

Byte Number	Type	Description
0–7	Character	This field contains one of the following: 1) An 8-character symbol name, padded with nulls 2) A pointer into the string table if the symbol name is longer than 8 characters
8–11	Long integer	Symbol value; storage class dependent
12–13	Short integer	Section number of the symbol
14–15	Unsigned short integer	Basic and derived type specification
16	Character	Storage class of the symbol
17	Character	Number of auxiliary entries (always 0 or 1)

A.7.1 Special Symbols

The symbol table contains some special symbols that are generated by the compiler, assembler, and linker. Each special symbol contains ordinary symbol table information as well as an auxiliary entry. Table A-10 lists these symbols.

Table A-10. Special Symbols in the Symbol Table

Symbol	Description
.file	File name
.text	Address of .text section
.data	Address of .data section
.bss	Address of .bss section
.bb	Address of the beginning of a block
.eb	Address of the end of a block
.bf	Address of the beginning of a function
.ef	Address of the end of a function
.target	Pointer to a structure or union that is returned by a function
.*n*fake	Dummy tag name for a structure, union, or enumeration
.eos	End of a structure, union, or enumeration
etext	Next available address after the end of the .text output section
edata	Next available address after the end of the .data output section
end	Next available address after the end of the .bss output section

Several of these symbols appear in pairs:

- .bb/.eb indicate the beginning and end of a block.
- .bf/.ef indicate the beginning and end of a function.
- .*n*fake/.eos name and define the limits of structures, unions, and enumerations that were not named. The .eos symbol is also paired with named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler assigns it a name so that it can be entered into the symbol table. These names are of the form .*n*fake, where *n* is an integer. The compiler begins numbering these symbol names at 0.

A.7.1.1 Symbols and Blocks

In C, a block is a compound statement that begins and ends with braces and contains symbol definitions. The symbol definitions for any particular block are grouped together in the symbol table, and are delineated by the .bb/.eb special symbols. Note that blocks can be nested in C, and their symbol table entries can also be nested correspondingly. Figure A-7 shows how block symbols are grouped in the symbol table.

	Symbol Table
Block 1:	.bb
	Symbols for block 1
	.eb
Block 2:	.bb
	Symbols for block 2
	.eb
	. .

Figure A-7. Symbols for Blocks

A.7.1.2 Symbols and Functions

The symbol definitions for a function appear in the symbol table as a group, delineated by .bf/.ef special symbols. The symbol table entry for the function name precedes the .bf special symbol. Figure A-8 shows the format of symbol table entries for a function.

Function Name
.bf
Symbols for the function
.ef

Figure A-8. Symbols for Functions

If a function returns a structure or union, then a symbol table entry for the special symbol .target will appear between the entries for the function name and the .bf special symbol.

A.7.2 Symbol Names

The first 8 bytes of a symbol table entry (bytes 0–7) indicate a symbol's name:

- If the symbol name is 8 characters or less, then 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 8 characters, then this field is treated as two long integers. The entire symbol name is stored in the string table. Bytes 0–3 contain 0, and bytes 4–7 are an offset into the string table.

A.7.3 String Table

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 instead contains 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 four.

Figure A-5 shows an example of 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 the Fourier_Transform.

40			
'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'	'\0'	'\0'

Figure A-9. Sample String Table

A.7.4 Storage Classes

Byte 16 of the symbol table entry indicates the storage class of the symbol. Storage classes refer to the method in which the C compiler accesses a symbol. Table A-11 lists valid storage classes.

Table A-11. Symbol Storage Classes

Mnemonic	Value	Storage Class	Mnemonic	Value	Storage Class
C_NULL	0	No storage class	C_UNTAG	12	Union tag
C_AUTO	1	Automatic variable	C_TPDEF	13	Type definition
C_EXT	2	External symbol	C_USTATIC	14	Uninitialized static
C_STAT	3	Static	C_ENTAG	15	Enumeration tag
C_REG	4	Register variable	C_MOE	16	Member of an enumeration
C_EXTDEF	5	External definition	C_REGPARM	17	Register parameter
C_LABEL	6	Label	C_FIELD	18	Bit field
C_ULABEL	7	Undefined label	C_BLOCK	100	Beginning or end of a block; used only for the .bb and .eb special symbols
C_MOS	8	Member of a structure	C_FCN	101	Beginning or end of a function; used only for the .bf and .ef special symbols
C_ARG	9	Function argument	C_EOS	102	End of structure; used only for the .eos special symbol
C_STRTAG	10	Structure tag	C_FILE	103	Filename; used only for the .file special symbol
C_MOU	11	Member of a union	C_LINE	104	Used only by utility programs

Some special symbols are restricted to certain storage classes. Table A-12 lists these symbols and their storage classes.

Table A-12. Special Symbols and Their Storage Classes

Special Symbol	Restricted to this Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

A.7.5 Symbol Values

Bytes 8–11 of a symbol table entry indicate a symbol's value. A symbol's value depends on the symbol's storage class; Table A-13 summarizes the storage classes and related values.

Table A-13. Symbol Values and Storage Classes

Storage Class	Value Description
C_AUTO	Stack offset in bits
C_EXT	Relocatable address
C_STAT	Relocatable address
C_REG	Register number
C_LABEL	Relocatable address
C_MOS	Offset in bits
C_ARG	Stack offset in bits
C_STRTAG	0
C_MOU	Offset in bits
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	Enumeration value
C_REGPARM	Register number
C_FIELD	Bit displacement
C_BLOCK	Relocatable address
C_FCN	Relocatable address
C_FILE	0

If a symbol's storage class is C_FILE, then the symbol's value is a pointer to the next .file symbol. Thus, the .file symbols form a one-way linked list in the symbol table. When there are no more .file symbols, the final .file symbol points back to the first .file symbol in the symbol table.

The value of a relocatable symbol is its virtual address. When the linker relocates a section, the value of a relocatable symbol changes accordingly.

A.7.6 Section Number

Bytes 12–13 of a symbol table entry contain a number that indicates which section the symbol was defined in. Table A-14 lists these numbers and the sections they indicate.

Table A-14. Section Numbers

Mnemonic	Section Number	Description
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1	.text section
N_SCNUM	2	.data section
N_SCNUM	3	.bss section
N_SCNUM	4–65,535	Section number of a named section, in the order in which the named sections are encountered

Note that if there were no .text, .data, or .bss sections, the numbering of named sections would begin with 1.

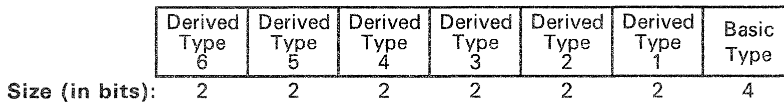
If a symbol has a section number of 0, -1, or -2, then it is not defined in a section. A section number of -2 indicates a symbolic debugging symbol, which includes structure, union, and enumeration tag names, type definitions, and filenames. A section number of -1 indicates that the symbol has a value but is not relocatable. A section number of 0 indicates a relocatable external symbol that is not defined in the current file.

A.7.7 Type Entry

Bytes 14–15 of the symbol table entry define the symbol's type. Each symbol symbol has:

- One basic type
- One to six derived types

The format for this 16-bit type entry is:



	Derived Type 6	Derived Type 5	Derived Type 4	Derived Type 3	Derived Type 2	Derived Type 1	Basic Type
Size (in bits):	2	2	2	2	2	2	4

Bits 0–3 of the type field indicate the basic type. Table A-15 lists valid basic types.

Table A-15. Basic Types

Mnemonic	Value	Type
T_NULL	0	Type not assigned
T_CHAR	2	Character
T_SHORT	3	Short integer
T_INT	4	Integer
T_LONG	5	Long integer
T_FLOAT	6	Floating point
T_DOUBLE	7	Double word
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_MOE	11	Member of an enumeration
T_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short integer
T_UINT	14	Unsigned integer
T_ULONG	15	Unsigned long integer

Bits 4–15 of the type field are arranged as six 2-bit fields which can indicate 1 to 6 derived types. Table A-16 lists the possible derived types.

Table A-16. Derived Types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARY	3	Array

An example of a symbol with several derived types would be a symbol with a type entry of 0000000011010011_2. This entry indicates that the symbol is a pointer to an array of short integers.

A.7.8 Auxiliary Entries

Each symbol table may have **one** or **no** auxiliary entry. An auxiliary symbol table entry contains the same number of bytes as a symbol table entry (18), but the format of an auxiliary entry depends on the symbol's type and storage class. Table A-17 summarizes these relationships.

Table A-17. Auxiliary Symbol Table Entries Format

Name	Storage Class	Type Entry		Auxiliary Entry Format
		Derived Type 1	Basic Type	
.file	C_FILE	DT_NON	T_NULL	Filename (see Table A-18)
.text, .data, .bss	C_STAT	DT_NON	T_NULL	Section (see Table A-19)
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	Tag name (see Table A-20)
.eos	C_EOS	DT_NON	T_NULL	End of structure (see Table A-21)
fcname	C_EXT C_STAT	DT_FCN	(See note 1)	Function (see Table A-22)
arrname	(See note 2)	DT_ARY	(See note 1)	Array (see Table A-23)
.bb, .eb	C_BLOCK	DT_NON	T_NULL	Beginning and end of a block (see Table A-24 and Table A-25)
.bf, .ef	C_FCN	DT_NON	T_NULL	Beginning and end of a function (see Table A-24 and Table A-25)
Name related to a structure, union or enumeration	(See note 2)	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION T_ENUM	Name related to a structure, union, or enumeration (see Table A-26)

Notes: 1) Any except T_MOE
2) C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF

In Table A-17, *tagname* refers to any symbol name (including the special symbol .*n*fake). *fcname* and *arrname* refer to any symbol name.

Any symbol that satisfies more than one condition in Table A-17 should have a union format in its auxiliary entry. Any symbol that does not satisfy any of these conditions should not have an auxiliary entry.

A.7.8.1 File Names

Each of the auxiliary table entries for a filename contains a 14-character file name in bytes 0–13. Bytes 14–17 are not used.

Table A-18. Section Format for Auxiliary Table Entries

Byte Number	Type	Description
0–13	Character	Filename
14–17	–	Not used

A.7.8.2 Sections

The auxiliary table entries for sections have the format shown in Table A-18.

Table A-19. Section Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Section length
4–6	Unsigned short integer	Number of relocation entries
7–8	Unsigned short integer	Number of line number entries
9–17	–	Not used (zero filled)

A.7.8.3 Tag Names

Table A-20 illustrates the format of auxiliary table entries for tag names.

Table A-20. Tag Name Format for Auxiliary Table Entries

Byte Number	Type	Description
0–5	–	Not used (zero filled)
6–7	Unsigned short integer	Size of structure, union, or enumeration
8–11	–	Not used (zero filled)
12–15	Long integer	Index of next entry beyond this structure, union, or enumeration
16–17	–	Not used (zero filled)

A.7.8.4 End of Structure

Table A-21 illustrates the format of auxiliary table entries for ends of structures.

Table A-21. End of Structure Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	–	Not used (zero filled)
6–7	Unsigned short integer	Size of structure, union, or enumeration
8–17	–	Not used (zero filled)

A.7.8.5 Functions

Table A-22 illustrates the format of auxiliary table entries for functions.

Table A-22. Function Format for Auxiliary Table Entries

Byte Number	Type	Description
0-3	Long integer	Tag index
4-7	Long integer	Size of function (in bits)
8-11	Long integer	File pointer to line number
12-15	Long integer	Index of next entry beyond this function
16-17	–	Not used (zero filled)

A.7.8.6 Arrays

Table A-23 illustrates the format of auxiliary table entries for arrays.

Table A-23. Array Format for Auxiliary Table Entries

Byte Number	Type	Description
0-3	Long integer	Tag index
4-5	Unsigned short integer	Line number declaration
6-7	Unsigned short integer	Size of array
8-9	Unsigned short integer	First dimension
10-11	Unsigned short integer	Second dimension
12-13	Unsigned short integer	Third dimension
14-15	Unsigned short integer	Fourth dimension
16-17	–	Not used (zero filled)

A.7.8.7 End of Blocks and Functions

Table A-24 illustrates the format of auxiliary table entries for the ends of blocks and functions.

Table A-24. End of Blocks and Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0-3	–	Not used (zero filled)
4-5	Unsigned short integer	C source line number
6-17	–	Not used (zero filled)

A.7.8.8 Beginning of Blocks and Functions

Table A-25 illustrates the format of auxiliary table entries for the beginnings of blocks and functions.

Table A-25. Beginning of Blocks and Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	–	Not used (zero filled)
4–5	Unsigned short integer	C source line number
6–11	–	Not used (zero filled)
12–15	Long integer	Index of next entry past this block
16–17	–	Not used (zero filled)

A.7.8.9 Names Related to Structures, Unions, and Enumerations

Table A-26 illustrates the format of auxiliary table entries for the names of structures, unions, and enumerations.

Table A-26. Structure, Union, and Enumeration Names Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	–	Not used (zero filled)
6–7	Unsigned short integer	Size of the structure, union, or enumeration
8–17	–	Not used (zero filled)
16–17	–	Not used (zero filled)

Appendix B

Symbolic Debugging Directives

The TMS320C30 assembler supports several directives that the TMS320C30 C compiler uses for symbolic debugging:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the symbol or function.
- The **.stag**, **.etag**, and **.utag** directives define structures, enumerations, and unions, respectively. The **.member** directive specifies a member of a structure, enumeration, or union. The **.eos** directive ends a structure, enumeration, or union definition.
- The **.func** and **.endfunc** directives specify the bounds of C blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source file name.
- The **.line** directive identifies the line number of a C source statement.

These symbolic debugging directives are not usually listed in the assembly language file that the compiler creates. If you want them to be listed, invoke the code generator with the -o option, as shown below:

```
cg30 -o <input file>
```

This appendix contains an alphabetical directory of the symbolic debugging directives. Each directive contains an example of C source and the resulting assembly language code.

Syntax **.block** *beginning line number*

.endblock *ending line number*

Description The .block and .endblock directives specify the beginning and end of a C block. The line numbers are optional; they specify the location in the source file where the block is defined.

Note that block definitions can be nested. The assembler will detect improper block nesting.

Example Here is an example of C source that defines a block, and the resulting assembly language code.

C source:

```
      .
      .
  {                       /*  Beginning of a block   */
        int  a,b;
        a = b;
  }                       /*  End of a block         */
      .
      .
      .
```

Resulting assembly language code:

```
        .block     0
        .sym       _a,1,4,1,32
        .sym       _b,2,4,1,32
        .line      7
        LDI        *+FP(2),R0
        STI        R0,*+FP(1)
        .endblock  7
```

Syntax **.file** *"filename"*

Description The .file directive allows a debugger to map locations in memory back to lines in a C source file. *Filename* is the name of the file that contains the orginal C source program. The first 14 characters of the filename are significant.

You can also use the .file directive in assembly code to provide a name in the file and improve program readability.

Example Here's an example of the .file directive. The file name named *text.c* contained the C source that produced this directive.

```
        .file    "text.c"
```

Syntax **.func** *beginning line number*

.endfunc *ending line number*

Description The .func and .endfunc directives specify the beginning and end of a C function. The line numbers are optional; they specify the location in the source file where the function is defined.

Note that function definitions cannot be nested.

Example Here is an example of C source that defines a function, and the resulting assembly language code.

C source:

```
power(x, n)               /*  Beginning of a function */
int x,n;
{
     int i, p;
     p = 1;
     for (i = 1; i <= n; ++i)
          p = p * x;
     return p;            /*  End of function          */
}
```

Resulting assembly language code:

```
0007                              .sym     _power,_power,36,2,0
0008                              .global  _power
0009
0010                              .func    2
0011                     ************************************
0012                      * FUNCTION DEF : _power
0013                     ************************************
0014 000000               _power:
0015 000000 0F2B0000              PUSH     FP
0016 000001 080B0014              LDI      SP,FP
0017 000002 02740001              ADDI     1,SP
0018 000003 0F240000              PUSH     R4
0019                              .sym     _x,-2,4,9,32
0020                              .sym     _n,-3,4,9,32
0021                              .sym     _i,1,4,1,32
0022                              .sym     _p,4,4,4,32
0023 000004                       .line    5
0024 000004 08640001              LDI      1,R4
0025 000005                       .line    6
0026 000005 15440301              STI      R4,*+FP(1)
0027 000006               L3:
0028 000006 08400301              LDI      *+FP(1),R0
0029 000007 04C00B03              CMPI     *+FP(3),R0
0030 000008 6A090008              BGT      L2
0031 000009                       .line    7
0032 000009 08000004              LDI      R4,R0
0033 00000A 08410B02              LDI      *-FP(2),R1
0034 00000B 62000000!             CALL     I_MULT
0035 00000C 08040000              LDI      R0,R4
0036 00000D 08410301              LDI      *+FP(1),R1
0037 00000E 02610001              ADDI     1,R1
0038 00000F 15410301              STI      R1,*+FP(1)
0039 000010 60000006+             BR       L3
0040 000011               L2:
0041 000011                       .line    8
0042 000011 08000004              LDI      R4,R0
0043 000012               EPIO_1:
0044 000012 0E240000              POP      R4
0045 000013 18740001              SUBI     1,SP
0046 000014 0E2B0000              POP      FP
0047 000015 78880000              RETS
0048
0049                              .endfunc         11
```

Syntax **line** *line number[,address]*

Description

The .line directive creates a line number entry in the object file. Line number entries are used in symbolic debugging to associate addresses in the object code with the lines in the source code that generated them.

The .line directive has two operands:

- *Line number* indicates the line of the C source that generated a portion of code. Line numbers are relative to the beginning of the current function. This is a required parameter.
- *Address* is an expression which is the address associated with the line number. This is an optional parameter; if you don't specify an address, the assembler will use the current SPC value.

Example

The .line directive is followed by the assembly language source statements that are generated by the indicated line of C source. For example, assume that the lines of C source below are line 4 and 5 in the original C source; lines 5 and 6 produce the assembly language source statements that are shown below.

C source:

```
for (i = 1; i <= n; ++i)
     p = p * x;
```

Resulting assembly language code:

```
0023 000004                      .line    5
0024 000004 08640001             LDI      1,R4
0025 000005                      .line    6
0026 000005 15440301             STI      R4,*+FP(1)
0027 000006           L3:
0028 000006 08400301             LDI      *+FP(1),R0
0029 000007 04C00B03             CMPI     *+FP(3),R0
0030 000008 6A090008             BGT      L2
0031 000009                      .line    7
0032 000009 08000004             LDI      R4,R0
0033 00000A 08410B02             LDI      *-FP(2),R1
0034 00000B 62000000!            CALL     I_MULT
0035 00000C 08040000             LDI      R0,R4
0036 00000D 08410301             LDI      *+FP(1),R1
0037 00000E 02610001             ADDI     1,R1
0038 00000F 15410301             STI      R1,*+FP(1)
0039 000010 60000006+            BR       L3
```

Syntax **.member** *name,value[,type,storage class,size,tag,dims]*

Description The .member directive defines a member of a structure, union, or enumeration. It is only valid when it appears in a structure, union, or enumeration definition.

- *Name* is the name of the member that is put in the symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the member. Any legal expression (absolute or relocatable) is acceptable.
- *Type* is the C type of the member. Appendix A contains more information about C types.
- *Storage class* is the C storage class of the member. Appendix A contains more information about C storage classes.
- *Size* is the number of bits of memory required to contain this member.
- *Tag* is the name of the type (if any) or structure of which this member is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be one to four expressions separated by commas. This allows up to four dimensions to be specified for the member.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example Here is an example of a C structure definition and the corresponding assembly language statements:

C source:

```
struct doc {
     char title;
     char group;
     int  job_number;
} doc_info;
```

Resulting assembly language code:

```
     .stag   doc,48
     .member _title,0,2,8,8
     .member _group,8,2,8,8
     .member _job_number,16,4,8,32
     .eos
```

Syntax

.stag *name[,size]*
member definitions
.eos

.etag *name[,size]*
member definitions
.eos

.utag *name[,size]*
member definitions
.eos

Description

The .stag directive begins a structure definition. The .etag directive begins an enumeration definition. The .utag directive begins a union definition. The .eos directive ends a structure, enumeration, or union definition.

- *Name* is the name of the structure, enumeration, or union. The first 32 characters of the name are significant. This is a required parameter.
- *Size* is the number of bits the structure, enumeration, or union occupies in memory. This is an optional parameter; if omitted, the size is unspecified.

The .stag, .etag, or.utag directive should be followed by a number of .member directives, which define members in the structure. The .member directive is the only directive that can appear inside a structure, enumeration, or union definition.

The assembler does not allow nested structures, enumerations, or unions. The C compiler "unwinds" nested structures by defining them separately and then referencing them from the structure they are referenced in.

Example 1

Here is an example of a structure definition.

C source:

```
struct doc
{
    char  title;
    char  group;
    int   job_number;
} doc_info;
```

Resulting assembly language code:

```
.stag   _doc,96
.member _title,0,2,8,32
.member _group,32,2,8,32
.member _job_number,64,4,8,32
.eos
```

Example 2 Here is an example of a union definition.

C source:

```
union u_tag {
    int   val1;
    float val2;
    char  valc;
} valu;
```

Resulting assembly language code:

```
        .utag   _u_tag,96
        .member _val1,0,2,8,32
        .member _val2,32,4,8,32
        .member _valc,64,4,8,32
        .eos
```

Example 3 Here is an example of an enumeration definition.

C Source:

```
{
    enum o_ty { reg_1, reg_2, result } optypes;
}
```

Resulting assembly language code:

```
        .etag   _o_ty,32
        .member _reg_1,0,11,16,32
        .member _reg_2,1,11,16,32
        .member _result,2,11,16,32
        .eos
```

Syntax **.sym** *name,value[,type,storage class,size,tag,dims]*

Description The .sym directive specifies symbolic debug information about a global variable, local variable, or a function.

- *Name* is the name of the variable that is put in the object symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the variable. Any legal expression (absolute or relocatable) is acceptable.
- *Type* is the C type of the variable. Appendix A contains more information about C types.
- *Storage class* is the C storage class of the variable. Appendix A contains more information about C storage classes.
- *Size* is the number of words of memory required to contain this variable.
- *Tag* is the name of the type (if any) or structure of which this variable is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be up to four expressions separated by commas. This allows up to four dimensions to be specified for the variable.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example These lines of C source produce the .sym directives shown below:

C source:

```
struct s { int member1, member2; } str;
int ext;
int array[5][10];
long *ptr;
int strcmp();

main(arg1,arg2)
   int arg1;
   char *arg2;
{
   register r1;
}
```

Resulting assembly language code:

```
      .sym   _str,_str,8,2,64,_s
      .sym   _ext,_ext,4,2,32
      .sym   _array,_array,244,2,1600,,5,10
      .sym   _ptr,_ptr,21,2,32
      .sym   _main,_main,36,2,0
      .sym   _arg1,_arg1,-2,4,9,32
      .sym   _arg2,_arg2,-3,18,9,32
      .sym   _r1,4,4,4,32
```

Appendix C

Assembler Error Messages

The assembler issues several types of error messages:

- Fatal,
- Nonfatal, **and**
- Macro errors.

When the assembler completes it second pass, it reports on any errors that it encountered during the assembly. It also prints these errors in the listing file (if one is created); an error is printed following the source line that incurred it.

This section discusses the three types of assembler error messages; they are listed in alphabetical order. Most errors are fatal errors; if an error is not fatal or if it is a macro error, this is noted in the list.

absolute value required: A relocatable symbol was used where an absolute symbol was expected.

address required: This instruction requires an address as an operand.

auxiliary register required for indirect: This instruction requires an auxiliary register as an operand.

blank missing: A blank or blanks must separate each field of the source statement.

cannot open library: A library name specified with the .mlib directive does not exist or is already being used.

closing (')') missing: Mismatched parenthesis.

closing quote missing: All strings must be enclosed in quotes.

comma missing: The assembler expected a comma but did not find one. This usually means that more operands were expected.

copy file open error: A file specified by a .copy directive does not exist or is already being used.

divide by zero: An expression or well-defined expression contains invalid division.

duplicate definition: The symbol appears as an operand of a REF statement, as well as in the the label field of the source, or, the symbol appears more than once in the label field of the source.

.else needs corresponding .if: An .else directive was not preceded by a .if directive.

$END statement missing in macro (macro error message): Within the macro library, an end of file was encountered before a $END card.

expression out of bounds:

expression syntax error: Unbalanced parentheses or invalid operations on relocatable symbols.

extended register required: This instruction requires an extended register (R0–R7) as an operand.

filename missing: The specified filename cannot be found.

floating-point number not valid in expression: Floating-point numbers cannot be used in expressions; you must use an integer instead.

$IF level exceeded (macro error message): The maximum nesting level of $IF directives is 44.

illegal label: A label cannot be used for the second instruction of a parallel instruction pair.

illegal structure, union, or enumeration tag

illegal structure definition

include/copy files not allowed in macro: You can't use the .copy directive within a macro.

incompatible addressing modes: An invalid combination of addressing modes has been used in an instruction.

incorrect macro definition (macro error message): Within the macro library, a macro was not found or a macro name was not given for a macro call.

index register required for displacement: Use an index register for indirect addressing.

indirect address required: This instruction expects an indirect address as an operand.

indirect displacement must be 0 or 1: The indirect placement for parallel instructions or three-operand instructions must be 0 or 1.

indirect displacement or out of bounds: The displacement for this instruction must be in the range 0–255.

invalid branch displacement: The specified displacement is a absolute but the SPC is relative, or the displacement is an external value, or the relative displacement is too large.

invalid bit-reversed modification

invalid circular modification

invalid expression: This may indicate invalid use of a relocatable symbol in arithmetic.

invalid floating-point constant:

invalid $IF structure (macro error message): The macro does not have matching $IF, $ELSE, and $ENDIF statements.

invalid $IF/$LOOP nesting (macro error message): An $IF used within a $LOOP must end within the $LOOP; a $LOOP within an $IF must end within the $IF.

invalid macro expansion (macro error message)

invalid macro library pathname (macro error message): The macro library name that was specified with an .mlib directive is invalid.

invalid macro qualifier (macro error message): The only valid macro qualifiers are S, V, L, A, SS, SV, SL, and SA.

invalid macro verb (macro error message)

invalid opcode: The command field of the source record has an entry that is not a defined instruction, directive, or macro name.

invalid option: An option specified by the .option directive is invalid.

invalid parallel instruction combination: The instructions specified as parallel instructions are not a valid pair.

invalid symbol: The symbol has invalid characters in it.

invalid symbol in macro expansion (macro error message)

invalid use of .asect

label required: The flagged directive must have a label.

long macro variable qualifier (macro error message): Macro variable qualifiers may be only one or two characters long.

library not archive: A file specified with an .mlib directive is not an archive file.

loop nesting level exceeded (macro error message)

macro line too long (macro error message): In a macro definition, macro directive lines may be only 53 characters long. Model statements, when fully expanded, may be only 55 characters long.

macro nesting level exceeded (macro error message)

missing first half of parallel instruction: The first instruction in a parallel instruction pair is missing or invalid.

operand missing: An operand must be supplied.

operand must be register or indirect: Three-operand and parallel instructions require register or indirect operands.

overflow in floating-point constant: Floating-point value to large to be represented.

pass1/pass2 operand conflict: A symbol in the symbol table did not have the same value in pass 1 and pass 2.

positive value required

R0 or R1 multiply destination required: The destination operand for an MPY||ADD pair or an MPY||SUB pair must be R0 or R1.

R2 or R3 ADD/SUB destination required: The destination operand for parallel ADD or SUB instructions must be R2 or R3.

register required: This instruction requires a register as an operand.

relocatable field must be 32 bits

string required: You must supply a string that is enclosed in double quotes.

symbol required: The .global directive requires a symbol as an operand.

symbol used in both REF and DEF: A REF symbol is already defined.

syntax error:

syntax error in macro assignment (macro error message)

syntax error in macro expansion (macro error message)

too many macro variables (macro error message): The total number of macro parameters, variables and labels in a single macro definition may not exceed 128.

unbalanced symbol table entries: For .block and .func directives.

undefined symbol: An undefined symbol was used where a well-defined expression is required.

underflow in floating-point constant: Floating-point value is too small to represent.

unexpected .endif encountered: An .endif directive was not preceded by an .if directive.

variable already defined (macro error message): A macro variable cannot be redefined within a macro.

warning - illegal relative branch: A branch has been requested to a different section.

warning - immediate operand not absolute

warning - null string defined: An empty string (one whose length = 0) is defined for string input, for directives that require a null string operand.

warning - register converted to immediate

warning - same destination registers: Parallel instructions must use different destination registers.

warning - symbol truncated: The maximum length for a symbol is eight characters. The assembler ignores the extra characters.

warning - trailing operand(s): The assembler found fewer or more operands than expected in the flagged instruction.

warning - value out of range

warning - value truncated: The expression given was to large to fit within the instruction opcode or the required number of bits.

Appendix D

Linker Error Messages

The linker issues several types of error messages:

- Syntax and command errors
- Allocation errors
- I/O errors

This section discusses the three types of errors; they are listed alphabetically within each category. The symbol "(...)" is used in these listings to represent the name of an object that the linker is attempting to interact with when an error occurs.

- ***Syntax/Command Errors***

  These errors are caused by incorrect use of linker directives, misuse of an input expression, invalid options, Check the syntax of all expressions, check the input directives for accuracy. Review the various options you are using and check for conflicts.

  absolute symbol (...) being redefined: An absolute symbol may not be redefined.

  adding name (...) to multiple output sections: The input section is mentioned twice in the SECTION directive.

  ALIGN illegal in this context: Alignment of a symbol may only be performed within a SECTIONS directive.

  attempt to decrement "."

  bad attribute value in MEMORY directive: (...): An attribute must be **R**, **W**, **X**, or **I**.

  bad flag value in SECTIONS directive, option (...)

  bad fill value: The fill value must be a 2-byte constant.

  binding excludes alignment: The section will be bound at the specified address regardless of the alignment of that address.

  both -r and -s flags are set; -s flag turned off: Since the -s option strips the relocation information and -r requests a relocatable object file, these options are in conflict with each other.

  -c requires fill value of 0 in .cinit: The value parameter has been overridden.

  -f flag does not specify a 2-byte number

  cannot align a section within GROUP – (...) not aligned

  cannot bind a section within a GROUP

cannot specify an owner for sections within a GROUP: The entire group is treated as one unit, so the group may be aligned or bound to an address, but the sections making up the group may not be handled individually.

cannot specify a page for a section within a GROUP

DSECT (...) can't be given an owner: Since dummy sections do not participate in memory allocation, it is meaningless for a dummy section to be given an owner or an attribute.

DSECT (...) can't be linked to an attribute

-e flag does not specify a legal symbol name (...)

entry point other than _c_int00 specified: For -c option only.

entry point symbol (...) undefined

errors in input - (...) not built

fill value on -f flag truncated to (...) bytes (warning)

ifile (comfile) nesting exceeded with file (...): Command file nesting is allowed up to 16 levels.

illegal operator in expression

misuse of "." symbol in assignment instruction: The dot symbol cannot be used in assignment statements that are outside SECTIONS directives.

no input files

number (...) not a power of 2: For the ALIGN operator.

-o file name too large (>128 char), truncated to (string)

-o flag does specify a valid file name : string

option flag does not specify a number

option is invalid flag

section (...) not built: The most likely cause of this is a syntax error in the SECTIONS directive.

semicolon required after expression

statement ignored: Caused by a syntax error in a expression.

symbol referencing errors - (...) not built

symbol (...) from file (...) being redefined: A defined symbol may not be redefined in an assignment statement.

syntax error: scanned line = (...)

unexpected EOF(end of file): Syntax error in the linker command file.

undefined symbol in expression

- *Allocation Errors*

These error messages appear during the allocation phase of linking. They generally appear if a section or group does not fit at a certain address or if the MEMORY and SECTION directives conflict in some way. If you are using a linker command file, check that MEMORY and SECTION directives allow enough room to ensure that no sections overlap and that no sections are being placed in unconfigured memory.

binding address (...) for section (...) is outside all memory on page (...)

binding address (...) for section (...) overlays previously allocated section

binding address (...) incompatible with alignment for section (...):

can't allocate output section, (...) of size (...) on page (...)

can't allocate section (...) with attribute (...) on page (...)

default allocation failed: (...) is too large

GROUP containing section (...) is too big

internal symbol (...) redefined in file (...): Ignored.

memory types (...) and (...) on page (...) overlap

no owner (...) for section (...) on page (...): Invalid or nonexistent memory range.

output file (...) not executable Warning.

PC-relative displacement overflow at address (...) in file (...)

section (...) at address (...) overlays previously allocated section (...) at address

section (...), bound at address (...), won't fit into page (...) of configured memory

section (...) enters unconfigured memory at address (...)

section (...) in file (...) is too big

undefined symbol (...) first referenced in file (...): Unless the -r option is used, the linker requires that all referenced symbols are defined.

- *I/O Errors:*

The following error messages indicate that the input file is corrupt, nonexistent, or unreadable or because the linker cannot write to the output file. Make sure that the input file is in the correct directory and that the file system is not out of space. If the input file is corrupt, try reassembling it.

cannot complete output file (...), write error

cannot create output file (...):

can't open (...)

can't read (...)

can't seek (...)

could not create map file (...)

fail to copy (...)

fail to read (...)

fail to seek (...)

fail to skip (...)

fail to write (...)

file (...) has no relocation information

file (...) is of unknown type, magic number = (...)

illegal relocation type (...) found in section(s) of file (...)

internal error : aux table overflow

invalid archive size for file (...)

I/O error on output file (...)

library (...) member has no relocation information

line number entry found for absolute symbol

memory allocation failure

no symbol map produced - not enough memory

relocation symbol not found: index (...), section (...), file (...)

relocation entries out of order in section (...) of file (...)

section (...) not found: An input section specified in a SECTIONS directive was not found in the input file.

sections .text, .data, or .bss not found: Optional header may be useless.

seek to (...) failed

Appendix E

ASCII Character Set

Base		Char	Base		Char	Base		Char	Base		Char
10	16		10	16		10	16		10	16	
0	00	NULL	32	29	SP	64	40	@	96	60	'
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	>
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	♂	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Appendix F

Glossary

absolute address: An address that is permanently assigned to a TMS320C30 memory location.

absolute section: An initialized named section defined with the .asect directive. All addresses in an absolute section are absolute.

alignment: A process in which the linker places an output section at an address that falls on an *n*-bit boundary, where *n* is a power of 2. You can specify alignment with the SECTIONS linker directive.

allocation: A process in which the linker determines 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 add new members.

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.

attribute component: Provides information about the origin and structure of a macro variable or macro symbol.

autoinitialization: The process of initializing global C variables (contained in the .cinit section) before beginning program execution.

auxiliary entry: A symbol may have an extra entry in the symbol table that contains additional information about the symbol (whether the symbol is a filename, a section name, a function name, etc.).

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: This is 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.

cache memory: A fast local memory onboard the TMS320C30. Blocks of code that are executed repeatedly can be loaded into the cache; this reduces the number of memory cycles and speeds program execution.

C compiler: A program that translates C source statements into TMS320C30 assembly language source statements.

command file: A file that contains linker options and names input files for the linker.

comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comment are not compiled, assembled, or linked; they have no effect on the object file.

common object file format (COFF): An object file that promotes modular programming by supporting the concept of *sections*.

conditional processing: A method of processing one block of source code or an alternate block of source code, based upon the evaluation of a specified expression.

configured memory: Memory that the linker has specified for allocation.

constant: A numeric value that can be used as an operand.

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

.data: This is one of the default COFF sections. The .data section is an initialized section that contains usually initialized data. You can use the .data directive to assemble code into the .data section.

digital signal processor: A microprocessor/microcomputer that performs algorithmic or numerical computational procedures upon digitized signals it has received and then sends the results to a host system or peripheral device.

digital signals: Digital representation of a continuous signal. Usually amplitude is represented at discrete time intervals with a digital value.

directive: Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

emulator: A hardware development system that emulates TMS320C30 operation.

entry point: The starting execution point in target memory.

enumeration:

executable module: An object file that has been linked and can be executed in a TMS320C30 system.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

external symbol: A symbol that is used in the current program module but defined in a different program module.

field: For the TMS320C30, 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).

global: Describes a symbol that is either 1) defined in the current module and accessed in another, or 2) accessed in the current module but defined in another.

GROUP: An option of the SECTIONS directive that forces specified output sections to be allocated contiguously (as a group).

high-level language debugging: The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

hole: An area between the input sections that comprise an output section which contains no actual code or data.

incremental linking: Linking files that have already been linked.

initialized section: A COFF section that contains executable code or initialized data. These sections can be built up with the .data, .text, .sect, or .asect directives.

input section: A section from an object file that will be linked into an executable module.

label: A symbol which begins in column 1 of a source statement and corresponds to the address of that statement.

length component: A component of a macro variable or macro symbol that contains the number of characters that make up the string.

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 TMS320C30 system memory and executed by the TMS320C30.

listing file: An output file created by the assembler that lists source statements, their line numbers, and their effects on the SPC.

loader: A device that loads an executable module into TMS320C30 system memory or to a debugging tool.

member: The elements or variables of a structure, union, or enumeration.

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; it's name must be the same as the macro name it defines, and it must have an extension of .asm.

macro variable: A variable that is valid within a macro definition or during a macro expansion.

magic number: An entry in the COFF file header that identifies an object file as a module that can be executed by the TMS320C30.

map file: An output file created by the linker that shows the memory configuration, section composition and allocation, and symbols and the addresses at which they were defined.

memory map: A map of TMS320C30 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

named section: A section that is defined with the .sect, .asect, or .usect directive. The .sect and .asect directives define *initialized* named sections that can be used like the .text and .data default sections. The .usect directive defines *uninitialized* named sections that can be used like the .bss default section.

object file: A file that has been assembled or linked and contains machine-language object code.

object format converter: A program that converts COFF object files into Intel-format or Tektronix-format object files.

object library: An archive library made up of individual object files.

operand: 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.

overlay pages: Multiple areas of physical memory that overlay each other at the same address. The TMS320C30 system can map different pages into the same address space in response to hardware select signals.

partial linking: Linking a file that will be linked again.

RAM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the -cr option. The RAM model allows variables to be initialized at load time instead of run time.

raw data: Executable code or initialized data in an output section.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

ROM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the -c option. In the ROM model, the linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

section: A relocatable block of code or data that will ultimately occupy contiguous space in the TMS320C30 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 TMS320C30 operation.

source file: A file that contains C code or TMS320C30 assembly language code that will compiled or assembled to form an object file.

SPC: Section Program Counter. An element of the assembler that keeps track of the current location within a section; each section has its own SPC.

static: Refers to a variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is re-entered.

storage class: Any entry in the symbol table that indicates how a symbol should be accessed.

string component: A copy of a string that is passed to a macro variable by a macro parameter or assigned to a macro symbol with an $ASG directive.

string table: Symbol names that are longer than 8 characters 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.

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.

tag: An optional "type" name that can be assigned to a structure, union, or enumeration.

target memory: Physical memory in a TMS320C30-based system into which executable object code will be loaded.

.text: This is 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.

unconfigured memory: Memory that is not defined as part of the TMS320C30 memory map and cannot be loaded with code or data.

uninitialized section: A COFF section that reserves space in the TMS320C30 memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

union: A variable which may hold (at different times) object of different types and sizes.

unsigned: Refers to a value that is treated as a positive number, regardless of its actual sign.

value component: A component of a macro variable or macro symbol that specifies the value of the variable or symbol.

well-defined expression: An expression that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

word: A 32-bit addressable location in target memory.

Index

A

a command (archiver) 8-3
-a option (linker) 9-4
absolute output module 9-4
absolute sections 3-6, 5-15
addressing modes 6-2
A—DIR (environment variable) 4-4, 4-5
.align (assembler directive) 5-14, 5-9
alignment 9-20
allocation 9-20, 9-27
alternate directories 9-7
    assembler 4-4-4-5
    linker 9-7
archive libraries 4-4, 5-36, 8-1-8-5, 9-7, 9-10, 9-13
archiver 1-3, 8-1-8-5
    examples 8-4
    in the development flow 1-2, 8-2
    invocation 8-3
    options 8-3
arithmetic instructions 6-22
arithmetic operators 4-12, 9-32
array definitions A-21
ASCII character set E-1
.asect (assembler directive) 5-15, 3-3, 3-4, 3-5, 3-6, 5-4, 5-41
assembler 1-3, 4-1-4-17
    character strings 4-11
    constants 4-8-4-10
    cross-reference listings 4-17
    directives 5-1-5-48
    error messages C-1-C-4
    expressions 4-12-4-14
    in the development flow 1-2, 4-2
    invocation 4-3
    macros 7-1-7-9
    output 4-15-4-17
    overview 4-1
    relocation 3-15
    sections 3-3-3-9
    source listings 4-15-4-16
    source statement format 4-6, 4-7
    symbols 3-17, 4-11
assembler directives 5-1, 5-48
    conditional assembly directives 5-11
        .else 5-32, 5-11
        .endif 5-32, 5-11
        .if 5-32, 5-11
    miscellaneous directives 5-23
        .end 5-23
    sections directives 3-3-3-9, 5-4
        .asect 5-15, 3-3-3-9, 5-4
        .bss 5-17, 3-3-3-9, 5-4
        .data 5-22, 3-3-3-9, 5-4
        .label 5-15, 5-4
        .sect 5-41, 3-3-3-9, 5-4, 5-15
        .text 5-45, 3-3-3-9, 5-4
        .usect 5-47, 3-3-3-9, 5-4
    summary table 5-2
    symbolic debugging directives B-1
        .block/.endblock B-1, B-2
        .etag/.eos B-1, B-8
        .file B-1, B-3
        .func/.endfunc B-1, B-4
        .line B-1, B-6
        .member B-1, B-7
        .stag/.eos B-1, B-8
        .sym B-1, B-10
        .utag/.eos B-1, B-8
    that align the SPC 5-9
        .align 5-14, 5-9
        .even 5-24, 5-9
        .field 5-9
    that format the output listing 5-10
        .length 5-34, 5-10
        .list 5-35, 5-10
        .mlist 5-38, 5-10
        .mnolist 5-38, 5-10
        .nolist 5-35, 5-10
        .option 5-39, 5-10
        .page 5-40, 5-10
        .title 5-46, 5-10
        .width 5-34, 5-10
    that initialize memory 5-6
        .byte 5-19, 5-6
        .field 5-25, 5-6
        .float 5-28, 5-6
        .hword 5-31, 5-6
        .int 5-33, 5-6
        .long 5-33, 5-6
        .set 5-42, 5-6
        .space 5-43, 5-6
        .string 5-44, 5-6
        .word 5-33, 5-6
    that reference other files 5-12
        .copy 5-20, 5-12
        .def 5-29, 5-12
        .global 5-29, 5-12
        .mlib 5-36, 5-12
        .ref 5-29, 5-12
assembler output 4-15-4-17, 5-10

assembly language development flow 1-2
assembly-time constants 4-10, 5-42
assigning a value to a symbol 5-42
autoinitialization 9-38, 9-40
  RAM model 9-6
  ROM model 9-6
auxiliary entries A-19

B

binary integers 4-8
binding 9-20
.block (assembler directive) B-1, B-2
block definitions A-13, A-21, A-22, B-2
.bss (assembler directive) 5-17, 3-3, 5-4
.bss section 3-3, 5-4, 5-17, 9-33, A-3
  holes 9-36
  initialization 9-36
.byte (assembler directive) 5-19, 5-6

C

C compiler 1-6, 9-6, 9-38-9-40, A-1, B-1-B-10
  block definitions B-2
  enumeration definitions B-8
  file identification B-3
  function definitions B-4
  line number entries A-9, B-6
  linking C code 9-6
  member definitions B-7
  special symbols A-12
  storage classes A-15
  structure definitions B-8
  symbol table entries B-10
  union definitions B-8
-c option (assembler) 4-3
-c option (linker) 9-6, 9-38-9-40
cache alignment 5-9, 5-14
C_DIR (environment variable) 9-7
character constants 4-9
character strings 4-11
characters 4-11, E-1
COFF 1-1, 3-1-3-17, 9-1, 10-1, A-1-A-22
  auxiliary entries A-19
  file headers A-4
  file structure A-2
  line number entries A-9, B-6
  relocation information A-8
  section headers A-6
  sections 3-1-3-17
  special symbols A-12
  string table A-14
  symbol table A-11
command files (linker) 9-3, 9-11
  example 9-42
comments (in source code) 4-7, 9-11
common object file format
  See COFF
compiler 1-3
condition codes (for the instruction set) 6-4
conditional blocks 5-11, 7-7
  assembler directives 5-11, 5-32
  macro directives 7-7
conditional expressions 4-13
configured memory 9-14, 9-27
constants 4-8
  assembly-time constants 4-10, 5-42
  binary integers 4-8
  characters 4-9
  decimal integers 4-9
  floating point 4-10, 5-28
  hexadecimal integers 4-9
  octal integers 4-8
.copy (assembler directive) 5-20, 4-4, 5-12
copy files 4-4
COPY section 9-29
-cr option (linker) 9-6, 9-38-9-40
cross-reference listings 4-17

D

d command (archiver) 8-3
.data (assembler directive) 5-22, 3-3, 3-4, 5-4
.data section 3-3, 5-4, 9-33, A-3
.datasection 5-22
decimal integers 4-9
.def (assembler directive) 5-29, 5-12
default fill value for holes 9-6
default sections 3-2, 5-22, 5-45
defining macros 7-4
development tools overview 1-2
directives
  See assembler directives
DSECT section 9-29
dummy section 9-29

E

e option (archiver) 8-3
-e option (linker) 9-6
.else (assembler directive) 5-32, 5-11
$ELSE (macro directive) 7-2, 7-7
emulator 1-3
.end (assembler directive) 5-23
.endblock (assembler directive) B-1, B-2
.endfunc (assembler directive) B-1, B-4
.endif (assembler directive) 5-32, 5-11
$ENDIF (macro directive) 7-2, 7-7
$ENDLOOP (macro directive) 7-2, 7-8
$ENDM (macro directive) 7-2
entry points for the linker 9-6
enumeration definitions B-8
enviroment variables 4-4
  A_DIR (assembler) 4-4
environment variables 9-8
  A_DIR (assembler) 4-5
  C_CIR 9-7
  C_DIR (linker) 9-7, 9-8
.eos (assembler directive) B-1, B-8
EPROM programmers 1-3, 10-1
error messages
  assembler C-1-C-4
  linker D-1-D-4
.etag (assembler directive) B-1, B-8
.even (assembler directive) 5-24, 5-9
expressions 4-12, 9-30
  conditional 4-13
  that are well defined 4-13
  that contain arithmetic operators 4-12
  that contain relocatable symbols 4-13
  underflow/underflow 4-13
external symbols 4-13, 5-12, 5-29, 5-42

F

-f option (linker) 9-6
.field (assembler directive) 5-25, 5-6
.file (assembler directive) 5-12, B-1, B-3
file headers A-4
file identification B-3
.float (assembler directive) 5-28, 5-6
floating point 4-10, 5-28
.func (assembler directive) B-1, B-4
function definitions A-13, A-21, A-22, B-4

G

.global (assembler directive) 5-29, 3-17, 5-12
global symbols 9-7
GROUP option (SECTIONS directive) 9-22

H

-h option (linker) 9-7
hexadecimal integers 4-9
hi-byte file 10-3
holes 9-6, 9-33
how to use this manual 1-5
.hword (assembler directive) 5-31, 5-6

I

-i option (assembler) 4-3, 4-4
-i option (linker) 9-7
-i option (object format converter) 10-3
.if (assembler directive) 5-32, 5-11
$IF (macro directive) 7-2, 7-7
.include (assembler directive) 4-4
include files 4-4
incremental linking 9-37
initialized sections 3-2, 3-4, 5-15, 5-22, 5-41, 5-45, 9-33
installation instructions 2-3
instruction set 1-6, 6-1-6-24
instructions 2-1
  PC-DOS 2-2
  VAX/VMS 2-3
.int (assembler directive) 5-33, 5-6
Intel object format 10-1, 10-3
interlocked-operation instructions 6-23
invoking the ...
  archiver 8-3
  assembler 1-4, 4-3
  linker 1-4, 9-3
  object format converter 10-3

L

-l option (assembler) 4-3
-l option (linker) 9-7
.label (assembler directive) 5-15, 5-4
labels 4-6
.length (assembler directive) 5-34, 5-10
.line (assembler directive) B-1, B-6
line number entries A-9, B-6

linker 1-3, 9-1-9-43
COFF 3-10-3-14, 9-1
command files 9-3, 9-11
command options summary 9-4
configured memory 9-14, 9-27
development flow 9-2
error messages D-1-D-4
example 9-41-9-43
expressions 9-30
in the development flow 1-2
invocation 9-3
linking C code 9-38-9-40
lnk30 command 9-3
loading a program 3-16
operators 9-32
relocation 3-15
sections 3-10-3-14
SECTIONS directive 9-16
symbols 3-17
unconfigured memory 9-14, 9-27
linker command files 9-3
linker command options 9-4-9-10
linking C code 9-6, 9-38-9-40
.list (assembler directive) 5-35, 5-10
listing control 5-35, 5-38, 5-39, 5-40, 5-46
listing file 4-15-4-16, 5-10
listing page size 5-34
lnk30 command 9-3
-a option 9-4
-c option 9-6, 9-38-9-40
-cr option 9-6, 9-38-9-40
-e option 9-6
-f option 9-6
-h option 9-7
-m option 9-9
-o option 9-10
options summary 9-4
-q option 9-10
-r option 9-4
-s option 9-10
-u option 9-10
load instructions 6-21
loading a program 3-16
lo-byte file 10-3
logical instructions 6-22
.long (assembler directive) 5-33, 5-6
$LOOP (macro directive) 7-2, 7-8

M

-m option (linker) 9-9
MACLIB files 5-36, 7-3
$MACRO (macro directive) 7-2, 7-4
macro libraries 4-4, 5-36, 7-3, 8-1
macros 7-1, 7-9
calls 7-1
conditional blocks 7-7
definitions 7-4
directives summary 7-2
MACLIB files 5-36, 7-3
macro libraries 5-36, 7-3
.mlib directive 5-36, 7-3
parameters 7-6
redefining opcodes 7-5
repeatable blocks 7-8
substitution 7-1
unique labels 7-9
manual organization 1-5
map file 9-9
example 9-43
.member (assembler directive) B-1, B-7
member definitions B-7
MEMORY (linker directive) 3-10, 9-14-9-15
default model 9-14
overlay pages 9-23
syntax 9-14
.mlib (assembler directive) 5-36, 4-4, 5-12, 7-3
.mlist (assembler directive) 5-38, 5-10
mnemonics 4-1
.mnolist (assembler directive) 5-38, 5-10

N

named memory 9-21
named sections 3-5, 3-2, 3-6, 5-4, 9-33, A-3
.asect 3-3, 3-5, 5-15
.sect 3-3, 3-5, 5-41
.usect 3-3, 3-5, 5-47
naming an output module 9-10
.nolist (assembler directive) 5-35, 5-10
NOLOAD section 9-29

O

-o option (linker) 9-10
object file format
    See COFF
object format converter 1-3, 10-1-10-4
    examples 10-4
    in the development flow 1-2, 10-2
    invocation 10-3
object libraries 8-1, 9-7, 9-13, 9-38
octal integers 4-8
operands 4-7
.option (assembler directive) 5-39, 5-10
optional file header A-5
output listing 4-15-4-16, 5-10
overflow (in expressions) 4-13
overlay pages 9-23-9-26

P

.page (assembler directive) 5-40, 5-10
parallel instructions 6-18
partially linked files 9-37
PC-DOS software installation 2-2
predefined symbols 4-11
program-control instructions 6-23

Q

q option (archiver) 8-3
-q option (assembler) 4-3
-q option (linker) 9-10

R

r command (archiver) 8-3
-r option (linker) 9-4, 9-37
RAM model (C compiler) 9-6, 9-38-9-40
redefining opcodes 7-5
.ref (assembler directive) 5-29, 5-12
related documentation 1-6
relinking 9-5, 9-10
    affected by -s 9-10
relocatable output module 9-5
relocatable symbols 4-13
relocation 3-15, 4-10, 9-4, 9-5, A-8
repeatable blocks 7-8
ROM model (C compiler) 9-6, 9-38-9-40
runtime initialization 9-38
runtime support 9-38

S

s option (archiver) 8-3
-s option (assembler) 4-3
-s option (linker) 9-10
.sect (assembler directive) 5-41, 3-3, 3-4, 3-5, 5-4, 5-15
section headers A-6
section specifications 9-17
sections 1-1, 3-1-3-17, 5-15, 5-45, 5-47
    default sections 3-2, 5-17, 5-22, 5-45
    named sections 3-2, 3-5, 5-15, 5-41, 5-47
SECTIONS (linker directive) 3-10, 9-16-9-22
    alignment 9-20
    allocation 9-20, 9-27
    binding 9-20
    default allocation 9-27
    GROUP option 9-22
    named memory 9-21
    overlay pages 9-24
    section specifications 9-17
    syntax 9-16
.set (assembler directive) 5-42, 5-6
simulator 1-3
software development system 1-3
software installation 2-1-2-3
    list of supported operating systems 2-1
    PC-DOS 2-2
    VAX/VMS 2-3
source listings 4-15-4-16
source statement format 4-6
    comment field 4-7
    label field 4-6
    mnemonic field 4-7
    operand field 4-7
    optional syntaxes 6-3
    parallel instructions 6-18
.space (assembler directive) 5-43, 5-6
SPC 3-6, 4-15, 9-34
    assembler symbol 4-7
    linker symbol 9-33
special section types 9-29
special symbols in the symbol table A-12
.stag (assembler directive) B-1, B-8
static symbols 9-7
static variables A-11
storage classes A-15
store instructions 6-21
.string (assembler directive) 5-44, 5-6
string table A-14
stripping line number entries 9-10
stripping symbolic information 9-10

structure definitions A-20, B-8
style and symbol conventions 1-7
support tools 1-1, 1-2
.sym (assembler directive) B-1, B-10
symbol names A-14
symbol table 3-17, A-11
symbol table entries B-10
symbolic debugging 9-10, A-9, A-11, B-1-B-10
    assembler directives 5-1, B-1
    block definitions B-2
    enumeration definitions B-8
    file identification B-3
    function definitions B-4
    line number entries B-6
    member definitions B-7
    -s assembler option 4-3
    structure definitions B-8
    symbol table entries B-10
    union definitions B-8
symbols 3-17, 4-11
    at link time 9-30
    character strings 4-11
    predefined 4-11
    relocatable symbols in expressions 4-13

T

t command (archiver) 8-3
-t option (object format converter) 10-3
Tektronix object format 10-1, 10-3
.text (assembler directive) 5-45, 3-3, 3-4, 5-4
.text section 3-3, 5-4, 5-45, 9-33, A-3
.title (assembler directive) 5-46, 5-10
TMS320C30 archiver
    See archiver
TMS320C30 assembler
    See assembler
TMS320C30 linker
    See linker
TMS320C30 object format converter
    See object format converter

U

-u option (linker) 9-10
unconfigured memory 9-14, 9-27
underflow (in expressions) 4-13
uninitialized sections 3-2, 3-3, 5-17, 5-47, 9-33
    holes 9-36
    initialization 9-36
union definitions B-8
unique labels for macros 7-9
.usect (assembler directive) 5-47, 3-3, 3-5, 5-4
.utag (assembler directive) B-1, B-8

V

v option (archiver) 8-3
VAX/VMS software installation 2-3

W

well-defined expressions 4-13
.width (assembler directive) 5-34, 5-10
.word (assembler directive) 5-33, 5-6

X

x command (archiver) 8-3
-x option (assembler) 4-3
-x option (object format converter) 10-3
XDS emulator 1-3

TI Worldwide Sales Offices

ALABAMA: Huntsville: 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Ave., Phoenix, AZ 85021, (602) 995-1007;**TUCSON:** 818 W. Miracle Mile, Suite 43, Tucson, AZ 85705, (602) 292-2640.

CALIFORNIA: Irvine: 17891 Cartwright Dr., Irvine, CA 92714, (714) 660-1200; **Roseville:** 1 Sierra Gate Plaza, Roseville, CA 95678, (916) 786-9208; **San Diego:** 4333 View Ridge Ave., Suite 100, San Diego, CA 92123, (619) 278-9601; **Santa Clara:** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Torrance:** 690 Knox St., Torrance, CA 90502, (213) 217-7010; **Woodland Hills:** 21220 Erwin St., Woodland Hills, CA 91367, (818) 704-7759.

COLORADO: Aurora: 1400 S. Potomac Ave., Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Altamonte Springs: 370 S. North Lake Blvd, Altamonte Springs, FL 32701, (305) 260-2116; **Ft. Lauderdale:** 2950 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Tampa:** 4803 George Rd., Suite 390, Tampa, FL 33634, (813) 885-7411.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7900

ILLINOIS: Arlington Heights: 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2925.

INDIANA: Ft. Wayne: 2020 Inwood Dr., Ft. Wayne, IN 46815, (219) 424-5174; **Carmel:** 550 Congressional Dr., Carmel, IN 46032, (317) 573-6400.

IOWA: Cedar Rapids: 373 Collins Rd. NE, Suite 201, Cedar Rapids, IA 52402, (319) 395-9550.

KANSAS: Overland Park: 7300 College Blvd., Lighton Plaza, Overland Park, KS 66210, (913) 451-4511.

MARYLAND: Columbia: 8815 Centre Park Dr., Columbia MD 21045, (301) 964-2003.

MASSACHUSETTS: Waltham: 950 Winter St., Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1569. **Grand Rapids:** 3075 Orchard Vista Dr. S.E., Grand Rapids, MI 49506, (616) 957-4200.

MINNESOTA: Eden Prairie: 11000 W. 78th St., Eden Prairie, MN 55344 (612) 828-9300.

MISSOURI: St. Louis: 11816 Borman Drive, St. Louis, MO 63146, (314) 569-7600.

NEW JERSEY: Iselin: 485E U.S. Route 1 South, Parkway Towers, Iselin, NJ 08830 (201) 750-1050.

NEW MEXICO: Albuquerque: 2820-D Broadbent Pkwy NE, Albuquerque, NM 87107, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Dr., East Syracuse, NY 13057, (315) 463-9291; **Melville:** 1895 Walt Whitman Rd., P.O. Box 2936, Melville, NY 11747, (516) 454-6600; **Pittsford:** 2851 Clover St., Pittsford, NY 14534, (716) 385-6770; **Poughkeepsie:** 385 South Rd., Poughkeepsie, NY 12601, (914) 473-2900.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0933; **Raleigh:** 2809 Highwoods Blvd., Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23775 Commerce Park Rd., Beachwood, OH 44122, (216) 464-6100; **Beavercreek:** 4200 Colonel Glenn Hwy., Beavercreek, OH 45431, (513) 427-6200.

OREGON: Beaverton: 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Blue Bell: 670 Sentry Pkwy, Blue Bell, PA 19422, (215) 825-9500.

PUERTO RICO: Hato Rey: Mercantil Plaza Bldg., Suite 505, Hato Rey, PR 00918, (809) 753-8700.

TENNESSEE: Johnson City: Erwin Hwy, P.O. Drawer 1255, Johnson City, TN 37605 (615) 461-2192.

TEXAS: Austin: 12501 Research Blvd., Austin, TX 78759, (512) 250-7655; **Richardson:** 1001 E. Campbell Rd., Richardson, TX 75081, (214) 680-5082; **Houston:** 9100 Southwest Frwy., Suite 250, Houston, TX 77074, (713) 778-6592; **San Antonio:** 1000 Central Parkway South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Murray: 5201 South Green St., Suite 200, Murray, UT 84123, (801) 266-8972.

WASHINGTON: Redmond: 5010 148th NE, Bldg B, Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Brookfield: 450 N. Sunny Slope, Suite 150, Brookfield, WI 53005, (414) 782-2899.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center, Nepean, Ontario, Canada, K2H9C4, (613) 726-1970. **Richmond Hill:** 280 Centre St. E., Richmond Hill L4C1B1, Ontario, Canada (416) 884-9181; **St. Laurent:** Ville St. Laurent Quebec, 9460 Trans Canada Hwy., St. Laurent, Quebec, Canada H4S1R7, (514) 336-1860.

ARGENTINA: Texas Instruments Argentina Viamonte 1119, 1053 Capital Federal, Buenos Aires, Argentina, 541/748-3699

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd.: 6-10 Talavera Rd., North Ryde (Sydney), New South Wales, Australia 2113, 2 + 887-1122; 5th Floor, 418 St. Kilda Road, Melbourne, Victoria, Australia 3004, 3 + 267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 8 + 255-2066.

AUSTRIA: Texas Instruments Ges.m.b.H.: Industriestrabe B/16, A-2345 Brunn/Gebirge, 2236-846210.

BELGIUM: Texas Instruments N.V. Belgium S.A.: 11, Avenue Jules Bondetlaan 11, 1140 Brussels, Belgium, (02) 242-3080.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda.: Rua Paes Leme, 524-7 Andar Pinheiros, 05424 Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Mairelundvej 46E, 2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND: Texas Instruments Finland OY: Ahertajantie 3, P.O. Box 81, ESPOO, Finland, (90) 0-461-422.

FRANCE: Texas Instruments France: Paris Office, BP 67 8-10 Avenue Morane-Saulnier, 78141 Velizy-Villacoublay cedex (1) 30 70 1003.

GERMANY (Fed. Republic of Germany): Texas Instruments Deutschland GmbH: Haggertystrasse 1, 8050 Freising, 8161+80-4591; Kurfuerstendamm 195/196, 1000 Berlin 15, 30+882-7365; III, Hagen 43/Kibbelstrasse, .19, 4300 Essen, 201-24250; Kirchhorsterstrasse 2, 3000 Hannover 51, 511+648021; Maybachstrabe 11, 7302 Ostfildern 2-Nelingen, 711+34030.

HONG KONG: Texas Instruments Hong Kong Ltd., 8th Floor, World Shipping Ctr., 7 Canton Rd., Kowloon, Hong Kong, (852) 3-7351223.

IRELAND: Texas Instruments (Ireland) Limited: 7/8 Harcourt Street, Stillorgan, County Dublin, Eire, 1 781677.

ITALY: Texas Instruments Italia S.p.A. Divisione Semiconduttori: Viale Europa, 40, 20093 Cologne Monzese (Milano), (02) 253001; Via Castello della Magliana, 38, 00148 Roma, (06) 5222651; Via Amendola, 17, 40100 Bologna, (051) 554004.

JAPAN: Tokyo Marketing/Sales (Headquarters): Texas Instruments Japan Ltd., MS Shibaura Bldg., 9F, 4-13-23 Shibaura, Minato-ku, Tokyo 108, Japan, 03-769-8700. Texas Instruments Japan Ltd.: Nissho-Iwai Bldg. 5F, 30 Imabashi 3-chome, Higashi-ku, Osaka 541, Japan, 06-294-1881; Daini Toyota West Bldg. 7F, 10-27 Meieki 4-chome, Nakamura-ku, Nagoya 450, 052-583-8691; Daiichi Seimei Bldg. 6F, 3-10 Oyama-cho, Kanazawa 920, Ishikawa-ken, 0762-23-5471; Daiichi Olympic Tachikawa Bldg. 6F, 1-25-12 Akebono-cho, Tachikawa 190, Tokyo, 0425-27-6426; Matsumoto Showa Bldg. 6F, 2-11 Fukashi 1-chome, Matsumoto 390, Nagano-ken, 0263-33-1060; Yokohama Nishiguchi KN Bldg. 6F, 2-8-4 Kita-Saiwai-cho, Nishi-ku, Yokohama 220, 045-322-6741; Nihon Seimei Kyoto Yasaka Bldg. 5F, 843-2 Higashi Shiokohjidori, Nishinotoh-in Higashi-iru, Shiokouji, Shimogyo-ku, Kyoto 600, 075-341-7713; 2597-1, Aza Harudai, Oaza Yasaka, Kitsuki 873, Oita-ken, 09786-3-3211; Miho Plant, 2350 Kihara Miho-mura, Inashiki-gun 300-04, Ibaragi-ken, 0298-85-2541.

KOREA: Texas Instruments Korea Ltd., 28th Fl., Trade Tower, #159, Samsung-Dong, Kangnam-ku, Seoul, Korea 2+551-2810.

MEXICO: Texas Instruments de Mexico S.A.: Alfonso Reyes—115, Col. Hipodromo Condesa, Mexico, D.F., Mexico 06120, 525/525-3860.

MIDDLE EAST: Texas Instruments: No. 13, 1st Floor Mannai Bldg., Diplomatic Area, P.O. Box 26335, Manama Bahrain, Arabian Gulf, 973+274681.

NETHERLANDS: Texas Instruments Holland B.V., 19 Hogehilweg, 1100 AZ Amsterdam—Zuidoost, Holland 20+5602911.

NORWAY: Texas Instruments Norway A/S: PB106, Refstad 0585, Oslo 5, Norway, (2) 155090.

PEOPLES REPUBLIC OF CHINA: Texas Instruments China Inc., Beijing Representative Office, 7-05 Citic Bldg., 19 Jianguomenwai Dajje, Beijing, China, (861) 5002255, Ext. 3750.

PHILIPPINES: Texas Instruments Asia Ltd.: 14th Floor, Ba- Lepanto Bldg., Paseo de Roxas, Makati, Metro Manila, Philippines, 817-60-31.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich, 2650 Moreira Da Maia, 4470 Maia, Portugal, 2-948-1003.

SINGAPORE (+ INDIA, INDONESIA, MALAYSIA, THAILAND): Texas Instruments Singapore (PTE) Ltd., Asia Pacific Division, 101 Thompson Rd. #23-01, United Square, Singapore 1130, 350-8100.

SPAIN: Texas Instruments Espana, S.A.: C/Jose Lazaro Galdiano No. 6, Madrid 28036, 1/458.14.58.

SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen): S-164-93, Stockholm, Sweden, 8 - 752-5800.

SWITZERLAND: Texas Instruments, Inc., Reidstrasse 6, CH-8953 Dietikon (Zuerich) Switzerland, 1-740 2220.

TAIWAN: Texas Instruments Supply Co., 9th Floor Bank Tower, 205 Tun Hwa N. Rd., Taipei, Taiwan, Republic of China, 2 + 713-9311.

UNITED KINGDOM: Texas Instruments Limited: Manton Lane, Bedford, MK41 7PA, England, 0234 270111.

TI Sales Offices

ALABAMA: Huntsville (205) 837-7530.

ARIZONA: Phoenix (602) 995-1007; Tucson (602) 292-2640.

CALIFORNIA: Irvine (714) 660-1200; Roseville (916) 786-9208; San Diego (619) 278-9601; Santa Clara (408) 980-9000; Torrance (213) 217-7010; Woodland Hills (818) 704-7759.

COLORADO: Aurora (303) 368-8000.

CONNECTICUT: Wallingford (203) 269-0074.

FLORIDA: Altamonte Springs (305) 260-2116; Ft. Lauderdale (305) 973-8502; Tampa (813) 885-7411.

GEORGIA: Norcross (404) 662-7900.

ILLINOIS: Arlington Heights (312) 640-2925.

INDIANA: Carmel (317) 573-6400; Ft. Wayne (219) 424-5174.

IOWA: Cedar Rapids (319) 395-9550.

KANSAS: Overland Park (913) 451-4511.

MARYLAND: Columbia (301) 964-2003.

MASSACHUSETTS: Waltham (617) 895-9100.

MICHIGAN: Farmington Hills (313) 553-1569; Grand Rapids (616) 957-4200.

MINNESOTA: Eden Prairie (612) 828-9300.

MISSOURI: St. Louis (314) 569-7600.

NEW JERSEY: Iselin (201) 750-1050.

NEW MEXICO: Albuquerque (505) 345-2555.

NEW YORK: East Syracuse (315) 463-9291; Melville (516) 454-6600; Pittsford (716) 385-6770; Poughkeepsie (914) 473-2900.

NORTH CAROLINA: Charlotte (704) 527-0933; Raleigh (919) 876-2725.

OHIO: Beachwood (216) 464-6100; Beaver Creek (513) 427-6200.

OREGON: Beaverton (503) 643-6758.

PENNSYLVANIA: Blue Bell (215) 825-9500.

PUERTO RICO: Hato Rey (809) 753-8700.

TENNESSEE: Johnson City (615) 461-2192.

TEXAS: Austin (512) 250-7655; Houston (713) 778-6592; Richardson (214) 680-5082; San Antonio (512) 496-1779.

UTAH: Murray (801) 266-8972.

WASHINGTON: Redmond (206) 881-3080.

WISCONSIN: Brookfield (414) 782-2899.

CANADA: Nepean, Ontario (613) 726-1970; Richmond Hill, Ontario (416) 884-9181; St. Laurent, Quebec (514) 336-1860.

TI Regional Technology Centers

CALIFORNIA: Irvine (714) 660-8105; Santa Clara (408) 748-2220;

GEORGIA: Norcross (404) 662-7945.

ILLINOIS Arlington Heights (312) 640-2909.

MASSACHUSETTS: Waltham (617) 895-9196.

TEXAS: Richardson (214) 680-5066.

CANADA: Nepean, Ontario (613) 726-1970.

TI Distributors

TI AUTHORIZED DISTRIBUTORS
Arrow/Kierulff Electronics Group
Arrow (Canada)
Future Electronics (Canada)
GRS Electronics Co., Inc.
Hall-Mark Electronics
Marshall Industries
Newark Electronics
Schweber Electronics
Time Electronics
Wyle Laboratories
Zeus Components

–OBSOLETE PRODUCT ONLY–
Rochester Electronics, Inc.
Newburyport, Massachusetts
(508) 462-9332

ALABAMA: Arrow/Kierulff (205) 837-6955; Hall-Mark (205) 837-8700; Marshall (205) 881-9235; Schweber (205) 895-0480.

ARIZONA: Arrow/Kierulff (602) 437-0750; Hall-Mark (602) 437-1200; Marshall (602) 496-0290; Schweber (602) 431-0030; Wyle (602) 866-2888.

CALIFORNIA: **Los Angeles/Orange County:** Arrow/Kierulff (818) 701-7500, (714) 838-5422; Hall-Mark (818) 773-4500, (714) 669-4100; Marshall (818) 407-0101, (818) 459-5500, (714) 458-5395; Schweber (818) 880-9686; (714) 863-0200, (213) 320-8090; Wyle (818) 880-9000, (714) 863-9953; Zeus (714) 921-9000; (818) 889-3838; **Sacramento:** Hall-Mark (916) 624-9781; Marshall (916) 635-9700; Schweber (916) 364-0222; Wyle (916) 638-5282; **San Diego:** Arrow/Kierulff (619) 565-4800; Hall-Mark (619) 268-1201; Marshall (619) 578-9600; Schweber (619) 450-0454; Wyle (619) 565-9171; **San Francisco Bay Area:** Arrow/Kierulff (408) 745-6600, Hall-Mark (408) 432-0900; Marshall (408) 942-4600; Schweber (408) 432-7171; Wyle (408) 727-2500; Zeus (408) 998-5121.

COLORADO: Arrow/Kierulff (303) 790-4444; Hall-Mark (303) 790-1662; Marshall (303) 451-8383; Schweber (303) 799-0258; Wyle (303) 457-9953.

CONNETICUT: Arrow/Kierulff (203) 265-7741; Hall-Mark (203) 271-2844; Marshall (203) 265-3822; Schweber (203) 264-4700.

FLORIDA: **Ft. Lauderdale:** Arrow/Kierulff (305) 429-8200; Hall-Mark (305) 971-9280; Marshall (305) 977-4880; Schweber (305) 977-7511; **Orlando:** Arrow/Kierulff (407) 323-0252; Hall-Mark (407) 830-5855; Marshall (407) 767-8585; Schweber (407) 331-7555; Zeus (407) 365-3000; **Tampa:** Hall-Mark (813) 530-4543; Marshall (813) 576-1399; Schweber (813) 541-5100.

GEORGIA: Arrow/Kierulff (404) 449-8252; Hall-Mark (404) 447-8000; Marshall (404) 923-5750; Schweber (404) 449-9170.

ILLINOIS: Arrow/Kierulff (312) 250-0500; Hall-Mark (312) 860-3800; Marshall (312) 490-0155; Newark (312) 784-5100; Schweber (312) 364-3750.

INDIANA: **Indianapolis:** Arrow/Kierulff (317) 243-9353; Hall-Mark (317) 872-8875; Marshall (317) 297-0483; Schweber (317) 843-1050.

IOWA: Arrow/Kierulff (319) 395-7230; Schweber (319) 373-1417.

KANSAS: **Kansas City:** Arrow/Kierulff (913) 541-9542; Hall-Mark (913) 888-4747; Marshall (913) 492-3121; Schweber (913) 492-2922.

MARYLAND: Arrow/Kierulff (301) 995-6002; Hall-Mark (301) 988-9800; Marshall (301) 235-9464; Schweber (301) 840-5900; Zeus (301) 997-1118.

MASSACHUSETTS Arrow/Kierulff (508) 658-0900; Hall-Mark (508) 667-0902; Marshall (508) 658-0810; Schweber (617) 275-5100; Time (617) 532-6200; Wyle (617) 273-7300; Zeus (617) 863-8800.

MICHIGAN: **Detroit:** Arrow/Kierulff (313) 462-2290; Hall-Mark (313) 462-1205; Marshall (313) 525-5850; Newark (313) 967-0600; Schweber (313) 525-8100; **Grand Rapids:** Arrow/Kierulff (616) 243-0912.

MINNESOTA: Arrow/Kierulff (612) 830-1800; Hall-Mark (612) 941-2600; Marshall (612) 559-2211; Schweber (612) 941-5280.

MISSOURI: **St. Louis:** Arrow/Kierulff (314) 567-6888; Hall-Mark (314) 291-5350; Marshall (314) 291-4650; Schweber (314) 739-0526.

NEW HAMPSHIRE: Arrow/Kierulff (603) 668-6968; Schweber (603) 625-2250.

NEW JERSEY: Arrow/Kierulff (201) 538-0900, (609) 596-8000; GRS Electronics (609) 964-8560; Hall-Mark (201) 575-4415, (201) 882-9773, (609) 235-1900; Marshall (201) 882-0320, (609) 234-9100; Schweber (201) 227-7880.

NEW MEXICO: Arrow/Kierulff (505) 243-4566.

NEW YORK: **Long Island:** Arrow/Kierulff (516) 231-1009; Hall-Mark (516) 737-0600; Marshall (516) 273-2424; Schweber (516) 334-7474; Zeus (914) 937-7400; **Rochester:** Arrow/Kierulff (716) 427-0300; Hall-Mark (716) 425-3300; Marshall (716) 235-7620; Schweber (716) 424-2222; **Syracuse:** Marshall (607) 798-1611.

NORTH CAROLINA: Arrow/Kierulff (919) 876-3132, (919) 725-8711; Hall-Mark (919) 872-0712; Marshall (919) 878-9882; Schweber (919) 876-0000.

OHIO: **Cleveland:** Arrow/Kierulff (216) 248-3990; Hall-Mark (216) 349-4632; Marshall (216) 248-1788; Schweber (216) 464-2970; **Columbus:** Hall-Mark (614) 888-3313; **Dayton:** Arrow/Kierulff (513) 435-5563; Marshall (513) 898-4480; Schweber (513) 439-1800.

OKLAHOMA: Arrow/Kierulff (918) 252-7537; Schweber (918) 622-8003.

OREGON: Arrow/Kierulff (503) 645-6456; Marshall (503) 644-5050; Wyle (503) 640-6000.

PENNSYLVANIA: Arrow/Kierulff (412) 856-7000, (215) 928-1800; GRS Electronics (215) 922-7037; Marshall (412) 963-0441; Schweber (215) 441-0600, (412) 963-6804.

TEXAS: **Austin:** Arrow/Kierulff (512) 835-4180; Hall-Mark (512) 258-8848; Marshall (512) 837-1991; Schweber (512) 339-0088; Wyle (512) 834-9957; **Dallas:** Arrow/Kierulff (214) 380-6464; Hall-Mark (214) 553-4300; Marshall (214) 233-5200; Schweber (214) 661-5010; Wyle (214) 235-9953; Zeus (214) 783-7010; **El Paso:** Marshall (915) 593-0706; **Houston:** Arrow/Kierulff (713) 530-4700; Hall-Mark (713) 781-6100; Marshall (713) 895-9200; Schweber (713) 784-3600; Wyle (713) 879-9953.

UTAH: Arrow/Kierulff (801) 973-6913; Hall-Mark (801) 972-1008; Marshall (801) 485-1551; Wyle (801) 974-9953.

WASHINGTON: Arrow/Kierulff (206) 575-4420; Marshall (206) 486-5747; Wyle (206) 881-1150.

WISCONSIN: Arrow/Kierulff (414) 792-0150; Hall-Mark (414) 797-7844; Marshall (414) 797-8400; Schweber (414) 784-9020.

CANADA: **Calgary:** Future (403) 235-5325; **Edmonton:** Future (403) 438-2858; **Montreal:** Arrow Canada (514) 735-5511; Future (514) 694-7710; **Ottawa:** Arrow Canada (613) 226-6903; Future (613) 820-8313; **Quebec City:** Arrow Canada (418) 871-7500; **Toronto:** Arrow Canada (416) 672-7769; Future (416) 638-4771; Marshall (416) 674-2161; **Vancouver:** Arrow Canada (604) 291-2986; Future (604) 294-1166.

Customer Response Center

TOLL FREE: (800) 232-3200

OUTSIDE USA: (214) 995-6611
(8:00 a.m. – 5:00 p.m. CST)

TEXAS INSTRUMENTS

Printed in U.S.A., November 1988
1604911-9705

SPRU035