TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide

Literature Number: SPRU024E
August 1999

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

Copyright © 1999, Texas Instruments Incorporated

Preface

Read This First

About This Manual

The *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide* tells you how to use these compiler tools:

- Compiler
- Source interlist utility
- Optimizer
- Parser
- Library-build utility

The TMS320C2x/C2xx/C5x C compiler accepts American National Standards Institute (ANSI) standard C source code and produces assembly language source code for the TMS320C2x/C2xx/C5x devices. This user's guide discusses the characteristics of the C compiler. It assumes that you already know how to write C programs. *The C Programming Language* (second edition), by Brian W. Kernighan and Dennis M. Ritchie (hereafter referred to as K & R), describes C based on the ANSI C standard. You can use the Kernighan and Ritchie book as a supplement to this manual.

Before you use the information about the C compiler in this User's Guide, you should read the *TMS320C1x/C2x/ C2xx/C5x Code Generation Tools Getting Started* to install the C compiler tools.

How to Use This Manual

The goal of this book is to help you learn how to use the Texas Instruments C compiler tools specifically designed for the TMS320C2x/C2xx/C5x devices. This book is divided into three distinct parts:

- **Introductory information**, consisting of Chapter 1, provides an overview of the TMS320C2x/C2xx/C5x development tools.
- **Compiler description**, consisting of Chapters 2, 5, 6, 7, and 8, describes how to operate the C compiler and the shell program, and discusses specific characteristics of the C compiler as they relate to the ANSI C specification. It contains technical information on the TMS320C2x/ C2xx/C5x architecture and includes information needed for interfacing assembly language to C programs. It describes libraries and header files in addition to the macros, functions, and types they declare. Finally, it describes the library-build utility.
- **Reference material**, consisting of Appendix B, provides a glossary.

Notational Conventions

This document uses the following conventions.

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

  Here is a sample of C code:

  ```
  # ifdef  NDEBUG
  # define assert
  ```

- In syntax descriptions, 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 must 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:

#include "*filename*"

  The **#include** preprocessor directive has one required parameter, *filename*. The filename must be enclosed in double quotes or angle brackets.

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not enter the brackets themselves. Here is an example of a command that has an optional parameter:

  clist *asmfile* [*outfile*] [*–options*]

  - The **clist** command has three parameters.
  - The first parameter, *asmfile*, is required.
  - The second and third parameters, *outfile* and *–options*, are optional.
  - If you omit the outfile, the file has the same name as the assembly file with the extension *.cl*.
  - Options are preceded by a hyphen.

Related Documentation From Texas Instruments

The following books describe the TMS320C2x/C2xx/C5x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, identify the book by its title and literature number (located on the bottom-right corner of the back cover).

TMS320C2x User's Guide (literature number SPRU014) discusses the hardware aspects of the 'C2x fixed-point digital signal processors. It describes pin assignments, architecture, instruction set, and software and hardware applications. It also includes electrical specifications and package mechanical data for all 'C2x devices. The book features a section with a 'C1x-to-'C2x DSP system migration.

TMS320C2xx User's Guide (literature number SPRU127) discusses the hardware aspects of the 'C2xx fixed-point digital signal processors. It describes pin assignments, architecture, instruction set, and software and hardware applications. It also includes electrical specifications and package mechanical data for all 'C2xx devices. The book features a section comparing instructions from 'C2x to 'C2xx.

TMS320C5x User's Guide (literature number SPRU056) describes the TMS320C5x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, instruction set, pipeline, specifications, DMA, and I/O ports. Software applications are covered in a dedicated chapter.

TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide (literature number SPRU018) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C1x, 'C2x, 'C2xx, and 'C5x generations of devices.

TMS320C2x Software Development System Technical Reference (literature number SPRU072) provides specific application and design information for using the TMS320C2x Software Development System (SWDS) board.

TMS320C5x Software Development System Technical Reference (literature number SPRU066) provides specific application and design information for using the TMS320C5x Software Development System (SWDS) board.

TMS320C2x C Source Debugger User's Guide (literature number SPRU070) tells how to use the 'C2x C source debugger with the 'C2x emulator, software development system (SWDS), and simulator.

TMS320C5x C Source Debugger User's Guide (literature number SPRU055) tells you how to invoke the 'C5x emulator, EVM, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints, and includes a tutorial that introduces basic debugger functionality.

Related Documentation

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

You may find these documents helpful as well:

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

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

Trademarks

MS-DOS is a registered trademark of Microsoft Corp.

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

SPARC is a trademark of SPARC International, Inc.

Sun-OS and Sun Workstation are trademarks of Sun Microsystems, Inc.

XDS is a trademark of Texas Instruments Incorporated.

Contents

1 Introduction **1-1**

Provides an overview of the TMS320C2x/C2xx/C5x software development tools, specifically, the optimizing C Compiler.

1.1 Software Development Tools Overview 1-2
1.2 C Compiler Overview 1-5

2 C Compiler Description **2-1**

Describes how to operate the C compiler and the shell program. Contains instructions for invoking the shell program, which compiles, assembles, and links a C source file. Discusses the interlist utility and compiler errors.

2.1 About the Shell Program 2-2
2.2 Invoking the Compiler Shell 2-4
2.3 Changing the Compiler's Behavior With Options 2-6
2.3.1 Frequently Used Options 2-13
2.3.2 Specifying Filenames 2-15
2.3.3 Changing How the Shell Program Interprets Filenames (–fa, –fc, and –fo Options) 2-15
2.3.4 Changing How the Shell Program Interprets and Names Extensions (–ea and –eo Options) 2-16
2.3.5 Specifying Directories 2-16
2.3.6 Options That Overlook ANSI C Type Checking 2-17
2.3.7 Runtime-Model Options 2-18
2.3.8 Options That Control the Assembler 2-19
2.4 Changing the Compiler's Behavior With Environment Variables 2-20
2.4.1 Setting Default Shell Options (C_OPTION) 2-20
2.4.2 Specifying a Temporary File Directory (TMP) 2-21
2.5 Controlling the Preprocessor 2-22
2.5.1 Predefined Macro Names 2-22
2.5.2 The Search Path for #include Files 2-23
2.5.3 Changing the #include File Search Path With the –i Option 2-24
2.5.4 Generating a Preprocessed Listing File (–pl Option) 2-25
2.5.5 Creating Custom Error Messages with the #error and #warn Directives 2-26
2.5.6 Enabling Trigraph Expansion (–p? Option) 2-26
2.5.7 Creating a Function Prototype Listing File (–pf Option) 2-26

2.6 Using Inline Function Expansion 2-27
2.6.1 Inlining Intrinsic Operators 2-27
2.6.2 Controlling Inline Function Expansion (–x Option) 2-28
2.6.3 Using the inline Keyword 2-28
2.6.4 The _INLINE Preprocessor Symbol 2-31
2.7 Using the Interlist Utility 2-33
2.8 Understanding and Handling Compiler Errors 2-35
2.8.1 Generating an Error Listing (–pr Option) 2-36
2.8.2 Treating Code-E Errors as Warnings (–pe Option) 2-36
2.8.3 Altering the Level of Warning Messages (–pw Option) 2-36
2.8.4 An Example of How You Can Use Error Options 2-37
2.9 Invoking the Tools Individually 2-38
2.9.1 Invoking the Parser 2-39
2.9.2 Parsing in Two Passes 2-41
2.9.3 Invoking the Optimizer 2-41
2.9.4 Invoking the Code Generator 2-43
2.9.5 Invoking the Interlist Utility 2-45

3 Optimizing Your Code 3-1

Describes how to optimize your C code, including such features as software pipelining and loop unrolling. Also describes the types of optimizations that are performed when you use the optimizer.

3.1 Using the C Compiler Optimizer 3-2
3.2 Using the –o3 Option 3-4
3.2.1 Controlling File-Level Optimization (–oln Option) 3-4
3.2.2 Creating an Optimization Information File (–onn Option) 3-5
3.3 Performing Program-Level Optimization (–pm and –o3 Options) 3-6
3.3.1 Controlling Program-Level Optimization (–opn Option) 3-6
3.3.2 Optimization Considerations When Mixing C and Assembly 3-8
3.3.3 Naming the Program Compilation Output File (–px Option) 3-9
3.4 Special Considerations When Using the Optimizer 3-10
3.4.1 Use Caution With asm Statements in Optimized Code 3-10
3.4.2 Use Caution With the Volatile Keyword 3-10
3.4.3 Use Caution When Accessing Aliased Variables 3-11
3.4.4 Assume Functions Are Not Interrupts 3-11
3.5 Automatic Inline Expansion (–oi Option) 3-12
3.6 Using the Interlist Utility With the Optimizer 3-13
3.7 Debugging Optimized Code 3-13
3.8 What Kind of Optimization Is Being Performed? 3-14
3.8.1 Cost-based Register Allocation 3-15
3.8.2 Autoincrement Addressing 3-15
3.8.3 Repeat Blocks 3-15
3.8.4 Delays, Banches, Calls, and Returns 3-16
3.8.5 Algebraic Reordering / Symbolic Simplification / Constant Folding 3-18
3.8.6 Alias Disambiguation 3-18

3.8.7 Data-Flow Optimizations . . . 3-18
3.8.8 Branch Optimizations and Control-Flow Simplification . . . 3-20
3.8.9 Loop Induction Variable Optimizations and Strength Reduction . . . 3-21
3.8.10 Loop Rotation . . . 3-21
3.8.11 Loop Invariant Code Motion . . . 3-21
3.8.12 Inline Expansion of Runtime-Support Library Functions . . . 3-21

4 Linking C Code . . . 4-1

Describes how to link using a standalone program or with the compiler shell and how to meet the special requirements of linking C code.

4.1 Invoking the Linker as an Individual Program . . . 4-2
4.2 Invoking the Linker With the Compiler Shell (–z Option) . . . 4-4
4.3 Disabling the Linker (–c Shell Option) . . . 4-5
4.4 Linker Options . . . 4-6
4.5 Controlling the Linking Process . . . 4-8
4.5.1 Linking With Runtime-Support Libraries . . . 4-8
4.5.2 Specifying the Type of Initialization . . . 4-9
4.5.3 Specifying Where to Allocate Sections in Memory . . . 4-11
4.5.4 A Sample Linker Command File . . . 4-13

5 TMS320C2x/C2xx/C5x C Language . . . 5-1

Discusses the specific characteristics of the TMS320C2x/C2xx/C5x C compiler as they relate to the ANSI C specification.

5.1 Characteristics of TMS320C2x/C2xx/C5x C Language . . . 5-2
5.1.1 Identifiers and Constants . . . 5-2
5.1.2 Data Types . . . 5-2
5.1.3 Conversions . . . 5-2
5.1.4 Expressions . . . 5-3
5.1.5 Declarations . . . 5-3
5.1.6 Preprocessor . . . 5-3
5.2 Data Types . . . 5-4
5.3 Register Variables . . . 5-6
5.4 Pragma Directives . . . 5-7
5.4.1 The CODE_SECTION Pragma . . . 5-7
5.4.2 The DATA_SECTION Pragma . . . 5-8
5.4.3 The FUNC_EXT_CALLED Pragma . . . 5-8
5.5 The asm Statement . . . 5-9
5.6 Creating Global Register Variables . . . 5-10
5.6.1 When to Use a Global Register Variable . . . 5-10
5.6.2 Avoiding Corrupting Register Values . . . 5-11
5.6.3 Disabling the Compiler From Using AR6 and AR7 . . . 5-11
5.7 Initializing Static and Global Variables . . . 5-12
5.7.1 Initializing Static and Global Variables With the const Type Qualifier . . . 5-12
5.7.2 Accessing I/O Port Space . . . 5-13
5.8 Compatibility with K&R C . . . 5-14
5.9 Compiler Limits . . . 5-16

6 Runtime Environment 6-1

Contains technical information on how the compiler uses the TMS320C2x/C2xx/C5x architecture. Discusses memory and register conventions, stack organization, function-call conventions, system initialization, and TMS320C2x/C2xx/C5x C compiler optimizations. Provides information needed for interfacing assembly language to C programs.

6.1 Memory Model 6-2
6.1.1 Sections 6-3
6.1.2 C System Stack 6-4
6.1.3 Allocating .const to Program Memory 6-5
6.1.4 Dynamic Memory Allocation 6-6
6.1.5 Initialization of Variables 6-7
6.1.6 Allocating Memory for Static and Global Variables 6-7
6.1.7 Field/Structure Alignment 6-8
6.1.8 Character String Constants 6-8
6.2 Register Conventions 6-9
6.2.1 Status Register Fields 6-11
6.2.2 Stack Pointer, Frame Pointer, and Local Variable Pointer 6-11
6.2.3 The TMS320C5x INDX Register 6-12
6.2.4 Register Variables 6-12
6.2.5 Expression Registers 6-13
6.2.6 Return Values 6-13
6.3 Function Structure and Calling Conventions 6-14
6.3.1 How a Function Makes a Call 6-15
6.3.2 How a Called Function Responds 6-15
6.3.3 Special Cases for a Called Function 6-16
6.3.4 Accessing Arguments and Local Variables 6-18
6.4 Interfacing C with Assembly Language 6-19
6.4.1 Using Assembly Language Modules With C Code 6-19
6.4.2 Using Inline Assembly Language 6-22
6.4.3 Accessing Assembly Language Variables From C Code 6-23
6.4.4 Modifying Compiler Output 6-24
6.5 Interrupt Handling 6-25
6.5.1 General Points About Interrupts 6-25
6.5.2 Using C Interrupt Routines 6-26
6.5.3 Using Assembly Language Interrupt Routines 6-27
6.5.4 TMS320C5x Shadow Register Capability 6-27
6.6 Integer Expression Analysis 6-28
6.6.1 Arithmetic Overflow and Underflow 6-28
6.6.2 Integer Division and Modulus 6-28
6.6.3 Long (32-Bit) Expression Analysis 6-28
6.6.4 C Code Access to the Upper 16 Bits of 16-Bit Multiply 6-29
6.7 Floating-Point Expression Analysis 6-30

6.8 System Initialization 6-31
6.8.1 Runtime Stack 6-32
6.8.2 Automatic Initialization of Variables 6-32
6.8.3 Initialization Tables 6-33
6.8.4 Autoinitialization of Variables at Runtime 6-34
6.8.5 Initialization of Variables at Load Time 6-35

7 Runtime-Support Functions 7-1

Describes the header files included with the C compiler, as well as the macros, functions, and types they declare. Summarizes the runtime-support functions according to category (header), and provides an alphabetical reference of the runtime-support functions.

7.1 Libraries 7-2
7.1.1 Linking Code With the Object Library 7-2
7.1.2 Modifying a Library Function 7-3
7.1.3 Building a Library With Different Options 7-3
7.2 Header Files 7-4
7.2.1 Diagnostic Messages (assert.h) 7-5
7.2.2 Character-Typing and Conversion (ctype.h) 7-5
7.2.3 Error Reporting (errno.h) 7-6
7.2.4 Limits (float.h and limits.h) 7-6
7.2.5 Inport/Outport Macros (ioports.h) 7-8
7.2.6 Floating-Point Math (math.h) 7-9
7.2.7 Nonlocal Jumps (setjmp.h) 7-9
7.2.8 Variable Arguments (stdarg.h) 7-9
7.2.9 Standard Definitions (stddef.h) 7-10
7.2.10 General Utilities (stdlib.h) 7-10
7.2.11 String Functions (string.h) 7-11
7.2.12 Time Functions (time.h) 7-11
7.3 Summary of Runtime-Support Functions and Macros 7-13
7.4 Description of Runtime-Support Functions and Macros 7-20

8 Library-Build Utility 8-1

Describes the utility that custom-makes runtime-support libraries for the options used to compile code. This utility can also be used to install header files in a directory and to create custom libraries from source archives.

8.1 Invoking the Library-Build Utility 8-2
8.2 Library-Build Utility Options 8-3
8.3 Options Summary 8-4

A Glossary A-1

Defines terms and acronyms used in this book.

Figures

1–1 TMS320C2x/C2xx/C5x Software Development Flow 1-2
2–1 The Shell Program Overview 2-3
2–2 Compiler Overview 2-38
3–1 Compiling a C Program With the Optimizer 3-2
6–1 Stack Use During a Function Call 6-14
6–2 Format of Initialization Records in the .cinit Section 6-33
6–3 Autoinitialization at Run time 6-34
6–4 Initialization at Load Time 6-35

Tables

2–1 Shell Options Summary 2-7
2–2 Predefined Macro Names 2-22
2–3 Example Error Messages 2-36
2–4 Selecting a Level for the –pw Option 2-37
2–5 Parser Options and dspcl Options 2-40
2–6 Optimizer Options and dspcl Options 2-42
2–7 Code Generator Options and dspcl Options 2-44
3–1 Options That You Can Use With –o3 3-4
3–2 Selecting a Level for the –ol Option 3-4
3–3 Selecting a Level for the –on Option 3-5
3–4 Selecting a Level for the –op Option 3-7
3–5 Special Considerations When Using the –op Option 3-7
4–1 Run-Time-Support Source Libraries 4-2
4–2 Sections Created by the Compiler 4-11
5–1 TMS320C2x/C2xx/C5x C Data Types 5-5
5–2 Absolute Compiler Limits 5-17
6–1 Register Use and Preservation Conventions 6-10
6–2 Status Register Fields 6-11
7–1 Macros That Supply Integer Type Range Limits (limits.h) 7-6
7–2 Macros That Supply Floating-Point Range Limits (float.h) 7-7
7–3 Summary of Run-Time-Support Functions and Macros 7-14
8–1 Summary of Options and Their Effects 8-4

Examples

2–1 How the Runtime-Support Library Uses the _INLINE Preprocessor Symbol 2-32
2–2 An Interlisted Assembly Language File 2-34
3–1 Repeat Blocks, Autoincrement Addressing, Parallel Instructions, Strength Reduction, Induction Variable Elimination, Register Variables, and Loop Test Replacement 3-16
3–2 Delayed Branch, Call, and Return Instructions 3-17
3–3 Data-Flow Optimizations 3-19
3–4 Copy Propagation and Control-Flow Simplification 3-20
3–5 Inline Function Expansion 3-22
4–1 An Example of a Linker Command File 4-13
5–1 Using the CODE_SECTION Pragma 5-7
5–2 Using the DATA_SECTION Pragma 5-8
6–1 TMS320C2x Code as a Called Function 6-16
6–2 An Assembly Language Function 6-21
6–3 Accessing a Variable Defined in .bss From C 6-23
6–4 Accessing a Variable Not Defined in .bss From C 6-24

Notes

Version Information 1-1
Function Inlining Can Greatly Increase Code Size 2-27
Using the –s Option With the Optimizer 2-34
Symbolic Debugging and Optimized Code 3-13
The _c_int0 Symbol 4-9
TMS320C2x/C2xx/C5x Byte Is 16 Bits 5-5
Avoid Disrupting the C Environment With asm Statements 5-9
The Linker Defines the Memory Map 6-2
Stack Overflow 6-5
Using AR6 and AR7 as Global Register Variables 6-13
Using the asm Statement 6-22
Customizing Time Functions 7-12
Writing Your Own Clock Function 7-27
Writing Your Own Time Function 7-57
TMS320C2x/C2xx/C5x Byte Is 16 Bits A-2

Chapter 1

Introduction

The TMS320C2x, TMS320C2xx, and TMS320C5x devices are members of the TMS320 family of high-performance CMOS microprocessors optimized for digital signal processing applications.

The TMS320C2x/C2xx/C5x DSPs are supported by a set of software development tools, which includes an optimizing C compiler, an assembler, a linker, an archiver, a software simulator, a full-speed emulator, and a software development board.

This chapter provides an overview of these tools and introduces the features of the optimizing C compiler. The assembler and linker are discussed in detail in the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

Note: Version Information

To use the TMS320C2x/C2xx/C5x C compiler, you must also have version 5.0 (or later) of the TMS320C1x/C2x/C2xx/C5x assembler and linker.

Topic **Page**

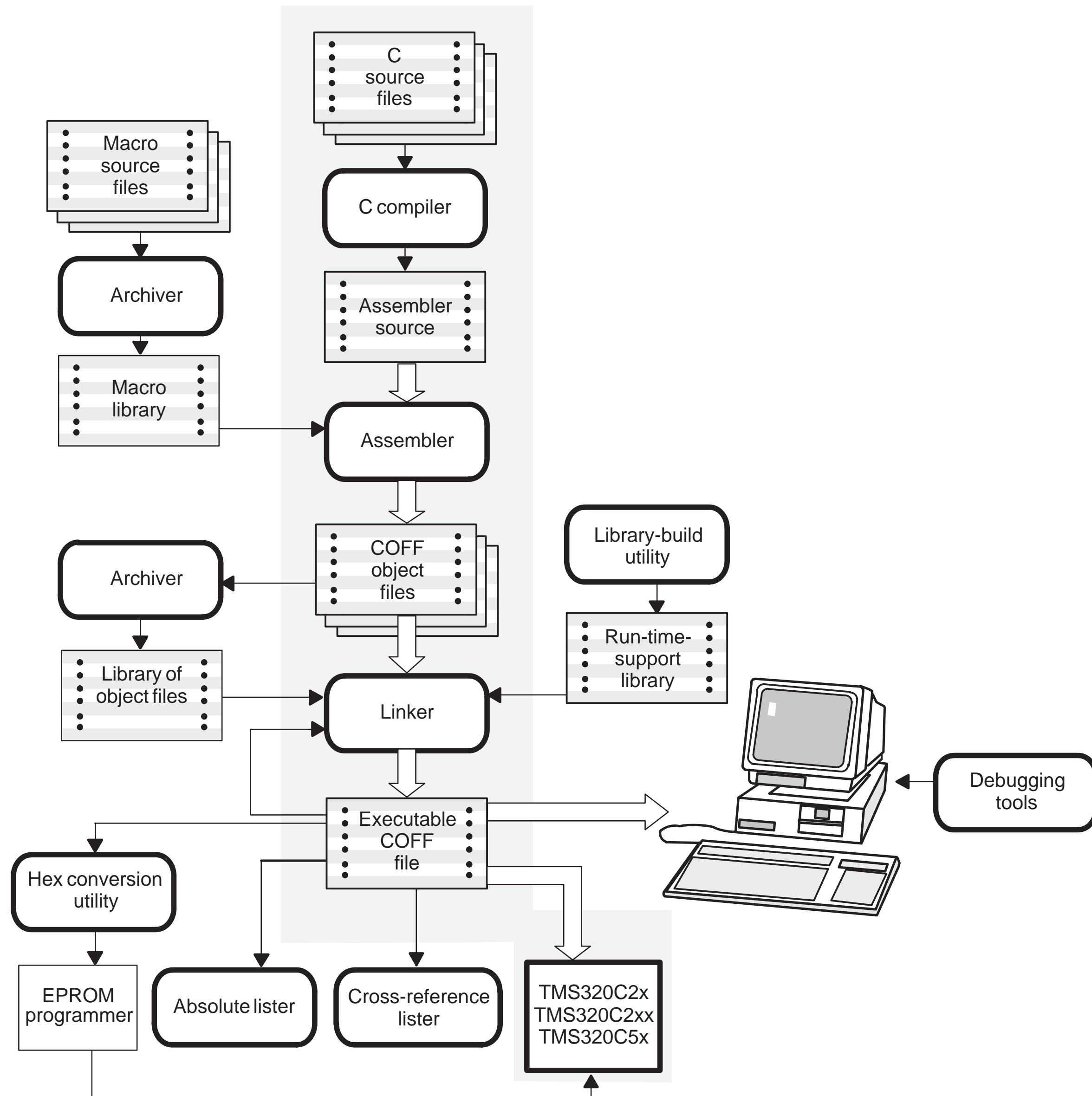
1.1 Software Development Tools Overview 1-2

1.2 C Compiler Overview 1-5

1.1 Software Development Tools Overview

Figure 1–1 illustrates the TMS320C2x/C2xx/C5x software development flow. The shaded portion of the figure highlights the most common path of software development for C language programs. The unshaded portions represent peripheral functions that enhance the development process.

Figure 1–1. TMS320C2x/C2xx/C5x Software Development Flow



The following list describes the tools that are shown in Figure 1–1.

- The **C compiler** accepts C source code and produces TMS320C2x, TMS320C2xx, or TMS320C5x assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package.
  - The **shell program** enables you to automatically compile, assemble, and link source modules.
  - The **optimizer** modifies code to improve the efficiency of C programs.
  - The **interlist utility** interlists C source statements with assembly language output.

Chapter 2 describes how to invoke and operate the compiler, the shell, the optimizer, and the interlist utility.

- The **assembler** translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). The *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide* explains how to use the assembler.

- The *TMS320C1x/C2x/C5x Assembly Language Tools User's Guide* explains how to use the archiver. 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 and object libraries as input.

- The **archiver** allows you to collect a group of files into a single archive file, called a *library*. Using the archiver, you can modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object modules. Three object libraries are shipped with the C compiler:
  - **rts25.lib** contains ANSI standard run-time-support functions and compiler-utility functions for the TMS320C2x.
  - **rts50.lib** contains ANSI standard run-time-support functions and compiler-utility functions for the TMS320C5x.
  - **rts2xx.lib** contains ANSI standard run-time-support functions and compiler-utility functions for the TMS320C2xx.

- The **library-build utility** allows you to build a customized run-time-support library. Standard run-time-support library functions are provided as source code. These are located in rts.src. See Chapter 8, *Library-Build Utility*, for more information.

- The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer.
- The **absolute lister** is a debugging tool. It accepts linked object files as input and creates .abs files as output. Once assembled, these .abs files produce lists that contain absolute rather than relative addresses. See the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide* for more information about how to use the absolute lister.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. See the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide* for more information about how to use the cross-reference lister.
- The main product of this development process is a module that you can execute on a **TMS320C2x/C2xx/C5x device**. You can use one of several debugging tools to refine and correct your code. Available products include:
  - An instruction-accurate software simulator
  - An extended development system (XDS-510) emulator
  - An evaluation module (EVM)

1.2 C Compiler Overview

The TMS320C2x/C2xx/C5x C compiler is a full-featured optimizing compiler that translates standard ANSI C programs into TMS320C2x/C2xx/C5x assembly language source. The following list describes key characteristics of the compiler:

- **ANSI-standard C**

  The TMS320C2x/C2xx/C5x compiler fully conforms to the ANSI C standard as defined by the ANSI specification and described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The ANSI C standard includes extensions to C that provide maximum portability and increased capability.

- **ANSI-standard run-time support**

  The compiler tools include a complete run time library for each device. All library functions conform to the ANSI C library standard. The libraries include functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, and exponential and hyperbolic functions. Functions for I/O and signal handling are not included because these are target-system specific. For more information, see Chapter 7, *Run-Time-Support Functions*.

- **Assembly source output**

  The compiler generates assembly language source files that you can inspect easily, enabling you to see the code generated from the C source files.

- **COFF object files**

  The common object file format (COFF) allows you to define your system's memory map at link time. This maximizes performance by enabling you to link C code and data objects into specific memory areas. COFF also supports source-level debugging.

- **Compiler shell program**

  The compiler tools include a shell program used to compile, and link programs in a single step. For more information, see section 2.1, *About the Shell Program,* on page 2-2.

- **Flexible assembly language interface**

  The compiler has clear calling conventions, allowing you to easily write assembly and C functions that call each other. For more information, see Chapter 6, *Run-Time Environment*.

- **Integrated preprocessor**

  The C preprocessor is integrated with the parser, allowing for faster compilation. Standalone preprocessing or preprocessed listing is also available. For more information, see section 2.5, *Controlling the Preprocessor*, on page 2-22.

- **Optimization**

  The compiler uses an optimization pass that employs several advanced techniques for generating efficient, compact code from C source. General optimizations can be applied to any C code, and TMS320C2x/C2xx/C5x-specific optimizations take advantage of the TMS320C2x/C2xx/C5x architecture. For more information about the C compiler's optimization techniques, see Chapter 3*, Optimizing Your Code*.

- **Code to initialized data into ROM**

  For standalone embedded applications, the compiler enables you to link all code and initialization data into ROM, allowing C code to run from reset.

- **Source interlist utility**

  The compiler tools include a utility that interlists your original C source statements into the assembly language output of the compiler. Using this utility, you can easily inspect the assembly code generated for each C statement. For more information, see section 2.7, *Using the Interlist Utility*, on page 2-33.

- **Library-build utility**

  The compiler tools include a utility that lets you easily custom-build object libraries from source for any combination of run time models or target CPUs. For more information, see Chapter 8, *Library-Build Utility*.

Chapter 2

C Compiler Description

Translating your source program into code that the TMS320C2x/C2xx/C5x can execute is a multistep process. You must compile, assemble, and link your source files to create an executable object file. The TMS320C2x/C2xx/C5x package contains a special shell program that enables you to execute all of these steps with one command. This chapter provides a complete description of how to use the dspcl shell to compile, assemble, and link your programs.

This chapter also describes the preprocessor, inline function expansion features, and interlist utility:

Topic **Page**

2.1 About the Shell Program 2-2

2.2 Invoking the Compiler Shell 2-4

2.3 Changing the Compiler's Behavior With Options 2-6

2.4 Changing the Compiler's Behavior With Environment Variables 2-20

2.5 Controlling the Preprocessor 2-22

2.6 Using Inline Function Expansion 2-27

2.7 Using the Interlist Utility 2-33

2.8 Understanding and Handling Compiler Errors 2-35

2.9 Invoking the Tools Individually 2-38

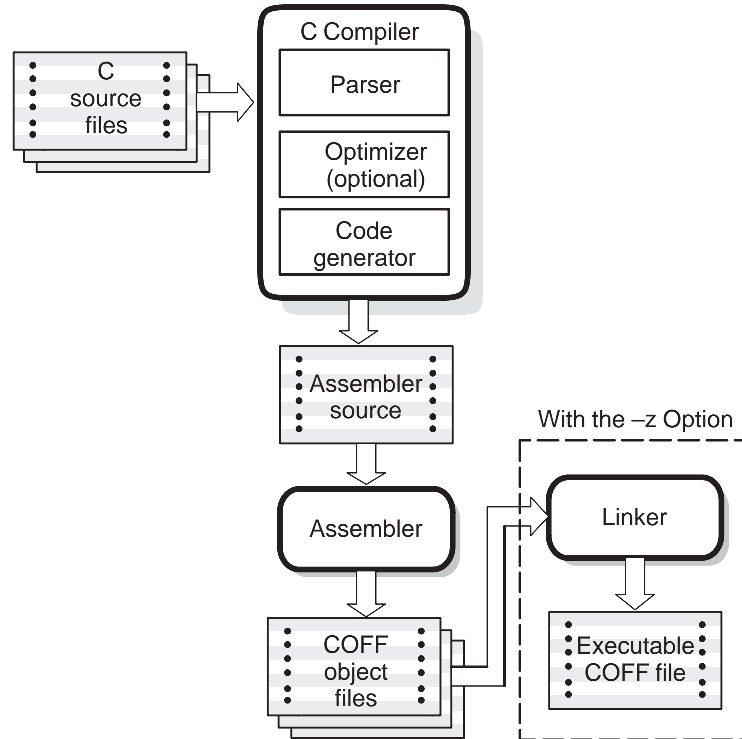
2.1 About the Shell Program

The compiler shell program (dspcl) lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the following:

- The **compiler**, which includes the parser, optimizer, and code generator, accepts C source code and produces ’C2x, ’C2xx, ’C5x, assembly language source code.
- The **assembler** generates a COFF object file.
- The **linker** links your files to create an executable object file. Use of the linker is optional at this point. You can compile and assemble various files with the shell and link them later. See Chapter 4, *Linking C Code,* for information about linking the files in a separate step.

The shell compiles and assembles files by default. If you use the –z option, dspcl will compile, assemble, and link your files. Figure 2–1 illustrates the paths the shell follows as it both uses and omits the linker.

Figure 2–1. The Shell Program Overview



For a complete description of the assembler and the linker, see the *TMS320C2x/C2xx/C5x Assembly Language Tools User's Guide*. For information about invoking the compiler tools individually, see section 2.9, *Invoking the Tools Individually,* on page 2-38.

2.2 Invoking the Compiler Shell

To invoke the compiler shell, enter:

dspcl [*options*] *filenames* [**–z**] [*linker options*] [*object files*]]

dspcl	is the command that runs the compiler and the assembler.
options	affect the way the shell processes input files.
filenames	are one or more C source files, assembly source files, or object files.
–z	is the option that runs the linker. See Chapter 4, *Linking C Code,* for more information about invoking the linker.
linker options	control the linking process.
object files	name the object files that the compiler creates.

The –z option and its associated information (linker options and object files) must follow all filenames and compiler options on the command line. You can specify all other options (except linker options) and filenames in any order on the command line. For example, if you wanted to compile two files named symtab.c and file.c, assemble a third file named seek.asm, and suppress progress messages (–q), you enter:

```
dspcl –q symtab file seek.asm
```

As dspcl encounters each source file, it prints the C filenames in square brackets ([ ]) and assembly language filenames in angle brackets (< >). This example uses the –q option to suppress the additional progress information that dspcl produces. Entering the command above produces these messages:

```
[symtab]
[file]
<seek.asm>
```

The normal progress information consists of a banner for each compiler pass and the names of functions as they are defined. The example below shows the output from compiling a single module *without* the –q option:

```
$ dspcl symtab
[symtab]
TMS320C2x/2xx/5x ANSI C Compiler        Version X.XX
Copyright (c) 1987–1995, Texas Instruments Incorporated
   ”symtab.c”: ==> main
   ”symtab.c”: ==> lookup
TMS320C2x/2xx/5x ANSI C Codegen      Version X.XX
Copyright (c) 1987–1995, Texas Instruments Incorporated
   ”symtab.c”: ==> main
   ”symtab.c”: ==> lookup
DSP Fixed Point COFF Assembler       Version X.XX
Copyright (c) 1987–1995, Texas Instruments Incorporated
   PASS 1
   PASS 2

No Errors, No Warnings
```

2.3 Changing the Compiler's Behavior With Options

Options control the operation of both the shell and the programs it runs. This section provides a description of option conventions and an option summary table. It also provides detailed descriptions of the most frequently used options, including options used for type-checking and assembling.

The following apply to the compiler options:

- Options are either single letters or two-letter pairs.
- Options are *not* case sensitive.
- Options are preceded by a hyphen.
- Single-letter options without parameters can be combined. For example, –sgq is equivalent to –s –g –q.
- Two-letter pair options that have the same first letter can be combined. For example, –pe, –pf, and –pk can be combined as –pefk.
- Options that have parameters, such as –u*name* and –i*directory*, cannot be combined. They must be specified separately.
- Options with parameters can have a space between the option and the parameter or be right next to each other.
- Files and options can occur in any order except the –z option. The –z option must follow all other compiler options and precede any linker options.

You can define default options for the shell by using the C_OPTION environment variable. For a detailed description of the C_OPTION environment variable, see section 2.4.1, *Setting Default Shell Options (C_OPTION)*, on page 2-20.

Table 2–1 summarizes all shell and linker options. Use the page references in the table to refer to more complete descriptions of the options.

For an online summary of all the shell and linker options, enter **dspcl** with no parameters on the command line.

Table 2–1. Shell Options Summary

(a) Options that control the compiler shell

Option	Effect	Page(s)
–@*filename*	Interprets contents of a file as one extension to the command line	2-13
–c	Disables linking (negate –z)	2-13, 4-5
–d *name*[**=***def*]	Predefines *name*	2-13
–g	Enables symbolic debugging	2-13
–i *directory*	Defines #include search path	2-13
–k	Keeps the assembly language (.asm) file	2-13
–n	Compiles only	2-13
–q	Suppresses progress messages (quiet)	2-13
–qq	Suppresses all messages (super quiet)	2-13
–r *<register>*	Reserves global register	2-13
–s	Interlists optimizer comments (if available) and assembly source statements; otherwise, interlists C and assembly source statements	2-14; 2-33
–ss	Interlists optimizer comments with C source and assembly statements	2-14
–u*name*	Undefines *name*	2-14
–v *xx*	Determines processor: *xx* = 25, 2xx or 50	2-14
–z	Enables linking	2-14

(b) Options that specify file and directory names

Option	Effect	Page
–fa*filename*	Identifies assembly language file (default for .asm or .s*)	2-15
–fc*filename*	Identifies C source file (default for .c or no extension)	2-15
–fo*filename*	Identifies object file (default for .o*)	2-15

Table 2–1. Shell Options Summary (Continued)

(c) Options that change the default file extensions

Option	Effect	Page
–ea*extension*	Sets default extension for assembly files	2-16
–eo*extension*	Sets default extension for object files	2-16

(d) Options that specify directories

Option	Effect	Page
–fr*directory*	Specifies object file directory	2-16
–fs*directory*	Specifies assembly file directory	2-16
–ft*directory*	Override TMP environment variable	2-16

(e) Options that overlook ANSI C type-checking

Option	Effect	Page
–tf	Relaxes prototype checking	2-17
–tp	Relaxes pointer combination	2-17

(f) Options that change the C run-time model

Option	Effect	Page
–ma	Assumes variables are aliased	2-18
–mb	Disables RPTK instruction	2-18
–ml	No LDPK optimization	2-18
–mn	Enables optimizations disabled by –g	2-18
–mp	Generates prolog/epilog inline	2-18
–mr	Lists register-use information	2-18
–ms	Optimizes for code space	2-18
–mx	Avoids ’C5x silicon bugs	2-18

Table 2–1. Shell Options Summary (Continued)

(g) Options that control the parser

Option	Effect	Page
–p?	Enables trigraph expansion	2-26
–pe	Treats code-E errors as warnings	2-36
–pf	Generates function prototype listing file	2-26
–pk	Allows K&R compatibility	
–pl	Generates preprocessed listing (.pp file)	2-25
–pm	Combines source files to perform program-level optimization	3-6
–pn	Suppresses #line directives in a .pp file	2-25
–po	Preprocesses only	2-25
–pr	Generates an error listing	2-36
–pw0	Disables all warning messages	2-36
–pw1	Enables serious warning messages (default)	2-36
–pw2	Enables all warning messages	2-36
–px*filename*	Names the output file created when using the –pm option	3-9

(h) Options that control the definition-controlled inline function expansion

Option	Effect	Page
–x0	Disables inline expansion of intrinsic operators	2-28
–x1	Enables inline expansion of intrinsic operators (default)	2-28
–x2 or –x	Defines the symbol _INLINE and invokes the optimizer with –o2	2-28

Table 2–1. Shell Options Summary (Continued)

(i) Options that control the assembler

Option	Effect	Page
–aa	Enables absolute listing	2-19
–ad*name*	Defines variables from the command line	2-19
–ahc*filename*	Copies *filename* before any source statements from the input file are assembled	2-19
–ahi*filename*	Includes *filename* before any source statements from the input file are assembled	2-19
–al	Generates an assembly listing file	2-19
–ap	Enables the 'C2x to 'C2xx or 'C5x port switch	2-19
–app	Enables the 'C2x to 'C2xx port switch and defines .TMS32025 and .TMS3202xx	2-19
–as	Puts labels in the symbol table	2-19
–au*name*	Undefines variables from the command line	2-19
–ax	Generates the cross-reference file	2-19

Table 2–1. Shell Options Summary (Continued)

(j) Options that control optimizations

Option	Effect	Page
–o0	Optimizes register usage	3-2
–o1	Uses –o0 optimizations and optimizes locally	3-2
–o2 or –o	Uses –o1 optimizations and optimizes globally	3-2
–o3	Uses –o2 optimizations and optimizes the file	3-3
–oe	Assumes no function in the module is called from an interrupt routine, and that none of the routines in the module are called recursively	3-11
–oi*size*	Sets automatic inlining size (–o3 only)	3-12
–ol0 (–oL0)	Informs the optimizer that your file alters a standard library function	3-4
–ol1 (–oL1)	Informs the optimizer that your file declares a standard library function	3-4
–ol2 (–oL2)	Informs the optimizer that your file does not declare or alter library functions. Overrides the –ol0 and –ol1 options (default).	3-4
–on0	Disables the optimization information file	3-5
–on1	Produces an optimization information file	3-5
–on2	Produces a verbose optimization information file	3-5
–op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	3-6
–op1	Specifies that the module contains variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code	3-6
–op2	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler (default)	3-6
–op3	Specifies that the module contains functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code	3-6
–os	Interlists optimizer comments with assembly statements	3-13

Table 2–1. Shell Options Summary (Continued)

(k) Options that control the linker

Options	Effect	Page
–a	Generates absolute executable output	4-6
–ar	Generates relocatable executable output	4-6
–b	Disables merge of symbolic debugging information	4-6
–c	Autoinitializes variables at run time	4-6
–cr	Initializes variables at reset	4-6
–e*global_symbol*	Defines entry point	4-6
–f*fill_value*	Defines fill value	4-6
–g*global_symbol*	Keeps *global_symbol* global (overrides –h)	4-6
–h	Makes global symbols static	4-6
–heap *size*	Sets heap size (bytes)	4-6
–i*directory*	Defines library search path	4-6
–l*filename*	Supplies library or command filename	4-6
–m*filename*	Names the map file	4-6
–n	Ignores all fill specifications in MEMORY directives	4-6
–o*filename*	Names the output file	4-7
–q	Suppresses progress messages (quiet)	4-7
–r	Generates relocatable nonexecutable output	4-7
–s	Strips symbol table information and line number entries from the output module	4-7
–stack *size*	Sets stack size (bytes)	4-7
–u*symbol*	Undefines symbol	4-7
–v0	Generates version 0 COFF format	4-7
–v1	Generates version 1 COFF format	4-7
–v2	Generates version 2 COFF format	4-7
–w	Displays a message when an undefined output section is created	4-7
–x	Forces rereading of libraries	4-7

2.3.1 Frequently Used Options

–@ *filename* — Appends the contents of a file to the command line. You can use this option to avoid limitations on command line length imposed by the host operating system. Use a # or j; at the beginning of a line in the command file to include comments.

–c — Suppresses the linker and overrides the –z option, which specifies linking. Use this option when you have –z specified in the C_OPTION environment variable and you do not want to link. For more information, see section 4.3, *Disabling the Linker (–c Shell Option)*, on page 4-5.

–d*name*[**=***def*] — Predefines the constant *name* for the preprocessor. This is equivalent to inserting #define *name def* at the top of each C source file. If the optional [=*def*] is omitted, the *name* is set to 1.

–g — Generates symbolic debugging directives that are used by the C-source-level debuggers and enables assembly source debugging in the assembler.

–i*directory* — Adds *directory* to the list of directories that the compiler searches for #include files. You can use this option a maximum of 32 times to define several directories. You must separate –i options with spaces. If you do not specify a directory name, the preprocessor ignores the –i option. For more information, see section 2.5.3, *Changing the #include File Search Path With the –i Option*, on page 2-24.

–k — Retains the assembly language output from the compiler. Normally, the shell deletes the output assembly language file after assembling completes.

–n — Compiles only. The specified source files are compiled but not assembled or linked. This option overrides –z. The output is assembly language output from the compiler.

–q — Suppresses banners and progress information from all the tools. Only source filenames and error messages are output.

–qq — Suppresses all output except error messages.

–r*register* — Reserves *register* globally so that the code generator and optimizer cannot use it as a normal save-on-entry register.

–s Invokes the interlist utility, which interweaves optimizer comments *or* C source with assembly source. If the optimizer is invoked (–o*n* option), optimizer comments are interlisted with the assembly language output of the compiler. If the optimizer is not invoked, C source statements are interlisted with the assembly language output of the compiler, which allows you to inspect the code generated for each C statement. The –s option implies the –k option.

–ss Invokes the interlist utility, which interweaves original C source with compiler-generated assembly language. If the optimizer is invoked (–o*n* option), this option might reorganize your code substantially. For more information, see section 2.7, *Using the Interlist Utility,* on page 2-33.

–u*name* Undefines the predefined constant *name*. This option overrides any –d options for the specified constant.

–v*xxx* Specifies the target processor. Choices for *xxx* are 25 for a ’C2x processor, 2xx for a ’C2xx, or 50 for a ’C5x.

–z Runs the linker on the specified object files. The –z option and its parameters follow all other options and parameters on the command line. All arguments that follow –z are passed to the linker. For more information, see section 4.1, *Invoking the Linker as an Individual Program,* on page 4-2.

2.3.2 Specifying Filenames

The input files that you specify on the command line can be C source files, assembly source files, or object files. The shell uses filename extensions to determine the file type.

Extension	File Type
.c or none (.c is assumed)	C source
.asm, .abs, or .s* (extension begins with s)	Assembly source
.obj	Object

Files without extensions are assumed to be C source files. The conventions for filename extensions allow you to compile C files and optimize and assemble assembly files with a single command.

For information about how you can alter the way that the shell interprets individual filenames, see section 2.3.3 on page 2-15. For information about how you can alter the way that the shell interprets and names the extensions of assembly source and object files, see section 2.3.5 on page 2-16.

You can use wildcard characters to compile or assemble multiple files. Wildcard specifications vary by system; use the appropriate form listed in your operating system manual. For example, to compile all of the C files in a directory, enter the following:

```
dspcl *.c
```

2.3.3 Changing How the Shell Program Interprets Filenames (–fa, –fc, and –fo Options)

You can use options to change how the shell interprets your filenames. If the extensions that you use are different from those recognized by the shell, you can use the –fa, –fc, and –fo options to specify the type of file. You can insert an optional space between the option and the filename. Select the appropriate option for the type of file you want to specify:

–fa*filename* for an assembly language source file

–fc*filename* for a C source file

–fo*filename* for an object file

For example, if you have a C source file called file.s and an assembly language source file called assy, use the –fa and –fc options to force the correct interpretation:

```
dspcl –fc file.s –fa assy
```

You cannot use the –fa, –fc, and –fo options with wildcard specifications.

2.3.4 Changing How the Shell Program Interprets and Names Extensions (–ea and –eo Options)

You can use options to change how the shell program interprets filename extensions and names the extensions of the files that it creates. The –ea and –eo options must precede the filenames they apply to on the command line. You can use wildcard specifications with these options. An extension can be up to nine characters in length. Select the appropriate option for the type of extension you want to specify:

–ea[**.**] *new extension*	for an assembly language file
–eo[**.**] *new extension*	for an object file

The following example assembles the file fit.rrr and creates an object file named fit.o:

```
dspcl –ea .rrr –eo .o fit.rrr
```

The period (.) in the extension and the space between the option and the extension are optional. You can also write the example above as:

```
dspcl –earrr –eoo fit.rrr
```

2.3.5 Specifying Directories

By default, the shell program places the object, assembly, and temporary files that it creates into the current directory. If you want the shell program to place these files in different directories, use the following options:

–fr*directory* Specifies a directory for object files. To specify an object file directory, type the directory's pathname on the command line after the –fr option:

```
dspcl –fr d:\object
```

–fs*directory* Specifies a directory for assembly files. To specify an assembly file directory, type the directory's pathname on the command line after the –fs option:

```
dspcl –fs d:\assembly
```

–ft*directory* Specifies a directory for temporary intermediate files. The –ft option overrides the TMP environment variable. (For more information, see section 2.4.2, *Specifying a Temporary File Directory (TMP)*, on page 2-21.) To specify a temporary directory, type the directory's pathname on the command line after the –ft option:

```
dspcl –ft c:\temp
```

2.3.6 Options That Overlook ANSI C Type Checking

Following are options that you can use to overlook some of the strict ANSI C type checking on your code:

–tf Overlooks type checking on redeclarations of prototyped functions. In ANSI C, if a function is declared with an old-format declaration and later declared with a prototype (as in the example below), this generates an error because the parameter types in the prototype disagree with the default argument promotions (which convert float to double and char to int).

```
int func( )                  /* old format */
int func(float a, char b)      /* new format */
```

–tp Overlooks type checking on pointer combinations. This option has two effects:

- A pointer to a signed type can be combined in an operation with a pointer to the corresponding unsigned type:

```
int *pi;
unsigned *pu;
pi = pu; /* Illegal unless -tp used */
```

- Pointers to differently qualified types can be combined:

```
char *p;
const char *pc;
p = pc;   /* Illegal unless -tp used */
```

The –tp option is especially useful when you pass pointers to prototyped functions, because the passed pointer type would ordinarily disagree with the declared parameter type in the prototype.

2.3.7 Run-Time-Model Options

–ma Assumes variables are aliased. The compiler assumes that pointers may alias (point to) named variables and aborts register optimizations when an assignment is made through a pointer.

–mb Disables the noninterruptible RPTK instruction for moving structures.

–ml Disables an optimization that the code generator performs to minimize the use of the LDPK instruction. This optimization can cause small holes in the .bss section of a program. Using the –ml option eliminates these holes entirely but at the expense of added LDPK instructions in the code. This could be a preferable tradeoff if your system uses a less expensive form of memory for program memory space than it does for data memory space.

–mn Reenables the optimizations disabled by **–g**. If you use the –g option to generate symbolic debugging information, many code generator optimizations are disabled because they disrupt the debugger.

–mr Lists register-use information. After the code generator compiles each C statement, –mr lists register content tables as comments in the assembly language file. The –mr option is useful for inspecting code that is difficult to follow due to register tracking optimizations.

–ms Optimizes for code space instead of for speed.

–mx Avoids 'C5x silicon bugs. Use of this switch is necessary when preparing a program for use with 'C5x device versions earlier than 2.0 that implements interrupts or is compiled with optimization.

When the compiler is run with the OVLY and RAM status bits on, certain compiled code sequences do not execute correctly when both the code and the data reside in the 1K of on-chip RAM on the 'C51 or the same 2K block of the 9K of on-chip RAM on the 'C50. Use a linker command file to set the program and data spaces so that this conflict does not occur.

2.3.8 Options That Control the Assembler

Following are assembler options that you can use with the shell:

–aa Invokes the assembler with the –a assembler option, which creates an absolute listing. An absolute listing shows the absolute addresses of the object code.

–ad*name* Invokes the assembler with the –hc assembler option to tell the assembler to copy the specified file for the assembly module. The file is inserted before source file statements. The copied file appears in the assembly listing files.

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

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

–al Invokes the assembler with the –l (lowercase L) assembler option to produce an assembly listing file.

–ap Enables ’C2x to ’C2xx or ’C5x port switch. Use –ap with the corresponding –v2xx or –v50 option.

–app Enables ’C2x to ’C2xx port switch and defines the .TMS32025 and .TMS3202xx assembler symbols. Use –app with the –v2xx option.

–as Invokes the assembler with the –s assembler option to put labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.

–au*name* Invokes the assembler with the –u assembler option to undefine the predefined constant *name*. The –au option overrides the –ad option for the specified constant.

–ax Invokes the assembler with the –x assembler option to produce a symbolic cross-reference in the listing file.

For more information about assembler options, see the *TMS320C6000 Assembly Language Tools User’s Guide.*

2.4 Changing the Compiler's Behavior With Environment Variables

You can define environment variables that set certain software tool parameters you normally use. An *environment variable* is a special system symbol that you define and associate to a string in your system initialization file. The compiler uses this symbol to find or obtain certain types of information.

When you use environment variables, default values are set, making each individual invocation of the compiler simpler because these parameters are automatically specified. When you invoke a tool, you can use command-line options to override many of the defaults that are set with environment variables.

2.4.1 Setting Default Shell Options (C_OPTION)

You might find it useful to set the compiler, assembler, and linker shell default options using the C_OPTION environment variable. If you do this, the shell uses the default options and/or input filenames that you name with C_OPTION every time you run the shell.

Setting the default options with the C_OPTION environment variable is useful when you want to run the shell consecutive times with the same set of options and/or input files. After the shell reads the command line and the input filenames, it looks for the C_OPTION environment variable and processes it.

The table below shows how to set the C_OPTION environment variable. Select the command for your operating system:

Operating System	Enter
DOS or OS/2	**set C_OPTION=***option$_1$*[;*option$_2$* . . .]
UNIX with C shell	**setenv C_OPTION "***option$_1$* [*option$_2$* . . .]**"**
UNIX with Bourne or Korn shell	**C_OPTION="***option$_1$* [*option$_2$* . . .]**"** **export C_OPTION**

Environment variable options are specified in the same way and have the same meaning as they do on the command line. For example, if you want to always run quietly (the –q option), enable C source interlisting (the –s option), and link (the –z option) for Windows, set up the C_OPTION environment variable as follows:

```
set C_OPTION=–qs –z
```

In the following examples, each time you run the compiler shell, it runs the linker. Any options following –z on the command line or in C_OPTION are passed to the linker. This enables you to use the C_OPTION environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the shell command line. If you have set –z in the environment variable and want to compile only, use the –c option of the shell. These additional examples assume C_OPTION is set as shown above:

```
dspcl *c                   ; compiles and links
dspcl –c *.c               ; only compiles
dspcl *.c –z lnk.cmd       ; compiles and links using a
                           ; command file
dspcl –c *.c –z lnk.cmd    ; only compiles (–c overrides –z)
```

For more information about shell options, see section 2.3, *Changing the Compiler's Behavior With Options*, on page 2-6. For more information about linker options, see section 4.4, *Linker Options*, on page 4-6.

2.4.2 Specifying a Temporary File Directory (TMP)

The compiler shell program creates intermediate files as it processes your program. By default, the shell puts intermediate files in the current directory. However, you can name a specific directory for temporary files by using the TMP environment variable.

Using the TMP environment variables allows use of a RAM disk or other file systems. It also allows source files to be compiled from a remote directory without writing any files into the directory where the source resides. This is useful for protected directories.

The table below shows how to set the TMP environment variable. Select the command for your operating system:

Operating System	Enter
DOS or OS/2	**set TMP=***pathname*
UNIX with C shell	**setenv TMP "***pathname***"**
UNIX with Bourne or Korn shell	**TMP="***pathname***"** **export TMP**

Note: For UNIX workstations, be sure to enclose the directory name within quotes.

For example, to set up a directory named temp for intermediate files on your hard drive for Windows, enter:

```
set TMP=c:\temp
```

2.5 Controlling the Preprocessor

During compilation, your code is run through the preprocessor, which is part of the parser. The shell program allows you to control the preprocessor with macros and various other preprocessor directives.

This section describes specific features that control the TMS320C2x/C2xx/C5x preprocessor. Refer to Section A12 of K&R for a general description of C preprocessing. The TMS320C2x/C2xx/C5x C compiler includes standard C preprocessing functions, which are built into the first pass of the compiler. The preprocessor handles the following:

- Macro definitions and expansions
- #include files
- Conditional compilation
- Various other preprocessor directives (specified in the source file as lines beginning with the # character)

The preprocessor produces self-explanatory error messages. The line number and the filename where the error occurred are printed along with a diagnostic message.

2.5.1 Predefined Macro Names

The compiler maintains and recognizes the predefined macro names listed in Table 2–2.

Table 2–2. Predefined Macro Names

Macro Name	Description
__LINE__ †	Expands to the current line number
__FILE__ †	Expands to the current source filename
__DATE__ †	Expands to the compilation date in the form *mm dd yyyy*
__TIME__ †	Expands to the compilation time in the form *hh:mm:ss*
_dsp	Expands to 1 (identifies the TMS320C2x/C2xx/C5x compiler)
_TMS320C25	Expands to 1 under the –v25 option
_TMS320C2XX	Expands to 1 under the –v2xx option
_TMS320C50	Expands to 1 under the –v50 option
_INLINE	Expands to 1 under the –x or –x2 option; undefined otherwise

† Specified by the ANSI standard

You can use the names listed in Table 2–2 the same manner as any other defined name. For example:

```
printf ( ”%s %s” , _ _TIME _ _ , _ _ DATE _ _);
```

translates to a line such as:

```
printf (”%s %s” , ”Jan 14 1988”, ”13:58:17”);
```

2.5.2 The Search Path for #include Files

The #include preprocessor directive tells the compiler to read source statements from another file. When specifying the file, you can enclose the filename in double quotes or in angle brackets. The filename can be a complete pathname, partial path information, or a filename with no path information.

- If you enclose the filename in double quotes (” ”), the compiler searches for the file in the following directories in this order:
  1) The directory that contains the current source file. The current source file refers to the file that is being compiled when the compiler encounters the #include directive.
  2) Directories named with the –i compiler
  3) Directories set with the C_DIR environment variable
- If you enclose the filename in angle brackets (< >), the compiler searches for the file in the following directories in this order:
  1) Directories named with the –i option
  2) Directories set with the C_DIR environment variable

See section 2.5.3, *Changing the Include File Search Path with the –i Option*, on page 2-24 for information on using the –i option. For information on how to use the C_DIR environment variable, see the *TMS320C1x/C2x/C2xx/C5x Code Generation Tools Getting Started Guide*.

2.5.3 Changing the #include File Search Path With the –i Option

The –i option names an alternative directory that contains #include files. The format of the –i option is:

–i *directory1* [**–i** *directory2* ...]

You can use up to 32 –i options per invocation of the compiler; each –i option names one *directory*. In C source, you can use the #include directive without specifying any directory information for the file; instead, you can specify the directory information with the –i option. For example, assume that a file called source.c is in the current directory. The file source.c contains the following directives statement:

```
#include "alt.h"
```

Assume that the complete pathname for alt.h is:

UNIX	/6xtools/files/alt.h
DOS or OS/2	c:\6xtools\files\alt.h

The table below shows how to invoke the compiler. Select the command for your operating system:

Operating System	Enter
DOS or OS/2	`dspcl –ic:\dsp\files source.c`
UNIX	`dspcl –i/dsp/files source.c`

2.5.4 Generating a Preprocessed Listing File (–pl Option)

The –pl option allows you to generate a preprocessed version of your source file with a .pp extension. The compiler's preprocessing functions perform the following on the source file:

- Each source line ending in backslash (\) is joined with the following line.
- Trigraph sequences are expanded (if enabled with the –p? option).
- Comments are removed.
- #include files are copied into the file.
- Macro definitions are processed.
- Macros are expanded.
- All other preprocessing directives, including #line directives and conditional compilation, are expanded.

These operations correspond to translation phases 1–3 specified in section A12 of K&R.

The preprocessed output file contains no preprocessor directives other than #line. The compiler inserts #line directives to synchronize line and file information in the output files with input position from the original source files. You can use the –pn option to suppress #line directives. See section 2.5.4.2, *Removing the #line Directives From the Preprocessed Listing File (–pn Option)* on page 2-25 for more information.

2.5.4.1 Generating a Preprocessed Listing File Without Code Generation (–po Option)

The –po option performs *only* the preprocessing functions and writes out the preprocessed listing file. The –po option is used instead of the –pl option. No syntax checking or code generation occurs. The –po option is useful when debugging macro definitions. The resulting listing file is a valid C source file that you can rerun through the compiler.

2.5.4.2 Removing the #line Directives From the Preprocessed Listing File (–pn Option)

The –pn option suppresses line and file information in the preprocessed listing file. The –pn option suppresses the #line directives in the .pp file generated with the –po or –pl option:

Here is an example of a #line directive:

```
#line 123 file.c
```

The –pn option is useful when compiling machine generated source.

2.5.5 Creating Custom Error Messages with the #error and #warn Directives

The standard #error preprocessor directive forces the compiler to issue a diagnostic message and halt compilation. The compiler extends the #error directive with a #warn directive. The #warn directive forces a diagnostic message but does not halt compilation. The syntax of #warn is identical to that of #error:

#error *token-sequence*

#warn *token-sequence*

2.5.6 Enabling Trigraph Expansion (–p? Option)

A trigraph is three characters that have a meaning (as defined by the ISO 646-1983 Invariant Code Set). On systems with limited character sets, these characters cannot be represented. For example, the trigraph ??' equates to ^. The ANSI C standard defines these sequences.

By default, the compiler does not recognize trigraphs. If you want to enable trigraph expansion, use the –pg option. During compilation, trigraphs are expanded to their corresponding single character. For more information about trigraphs, see the ANSI specification, § 2.2.1.1.

2.5.7 Creating a Function Prototype Listing File (–pf Option)

When you use the –pf option, the preprocessor creates a file containing the prototype of every function in all corresponding C files. Each function prototype file is named as its corresponding C file with a .pro extension.

2.6 Using Inline Function Expansion

When an inline function is called, the C source code for the function is inserted at the point of the call. This is known as inline function expansion. Inline function expansion is advantageous in short functions for the following reasons:

- It saves the overhead of a function call.
- Once inlined, the optimizer is free to optimize the function in context with the surrounding code.

Inline function expansion is performed in one of the following ways:

- Intrinsic operators are expanded by default.
- Automatic inline expansion is performed on small functions that are invoked by the optimizer with the –o3 option. For more information, see section 3.5, *Automatic Inline Expansion (–oi Option)*, on page 3-12.
- Definition-controlled inline expansion is performed when you invoke the compiler with optimization (–x option) and the compiler encounters the inline keyword in code.

Note: Function Inlining Can Greatly Increase Code Size

Expanding functions inline expands code size, and inlining a function that is called in a number of places increases code size. Function inlining is optimal for functions that are called only from a small number of places and for small functions. If your code size seems too large, try compiling with the –oi0 option and note the difference in code size.

2.6.1 Inlining Intrinsic Operators

The compiler automatically expands the intrinsic operators of the target system (such as abs) by default. This expansion happens whether or not you use the optimizer and whether or not you use any compiler or optimizer options on the command line. (You can defeat this automatic inlining by invoking the compiler with the –x0 option.) Functions that expand the intrinsic operators are:

- abs
- labs
- fabs

2.6.2 Controlling Inline Function Expansion (–x Option)

The –x option controls the definition of the _INLINE preprocessor symbol and some types of inline function expansion. There are three levels of expansion:

–x0 Causes no definition-controlled inline expansion. This option overrides the default expansions of the intrinsic operator functions, but it does not override the inline function expansions described in section 3.5, *Automatic Inline Expansion (–oisize Option)*, on page 3-12.

–x1 Resets the default behavior. The intrinsic operators (abs, labs, and fabs) are inlined wherever they are called. Use this option to reset the default behavior from the command line if you have used another –x option in an environment variable or command file.

–x2 or **–x** Defines the _INLINE preprocessor symbol to be 1. If the optimizer is not invoked with a separate command-line option, this option invokes the optimizer at the default level (–o2).

2.6.3 Using the inline Keyword

Definition-controlled inline expansion is performed when you invoke the compiler with optimization and the compiler encounters the inline keyword in code. Functions with local static variables or a variable number of arguments are not inlined, with the exception of functions declared as static inline. In functions declared as static inline, expansion occurs despite the presence of local static variables. In addition, a limit is placed on the depth of inlining for recursive or nonleaf functions. Inlining should be used for small functions or functions that are called in a few places (though the compiler does not enforce this). You can control this type of function inlining with the inline keyword.

The inline keyword specifies that a function is expanded inline at the point at which it is called rather than by using standard calling procedures. The compiler performs inline expansion of functions declared with the inline keyword, and can automatically inline small functions.

For a function to be eligible for inlining:

- The function must be declared with the inline keyword, or
- The optimizer must be invoked using the –o3 switch, and
  - The function is very small (controlled by the –oi switch), and
  - The function is declared before it is called

A function may be disqualified from inlining if it:

- Returns a struct or union
- Has a struct or union parameter
- Has a volatile parameter
- Has a variable length argument list
- Declares a struct, union, or enum type
- Contains a static variable
- Contains a volatile variable
- Is recursive
- Contains # pragmas
- Has too large of a stack (too many local variables)

2.6.3.1 Declaring a Function as Inline Within a Module

By declaring a function as inline within a module (with the inline keyword), you can specify that the function is inlined within that module. A global symbol for the function is created (code is generated), but the function is inlined only within the module where it is declared as inline. The global symbol can be called by other modules if they do not contain a static inline declaration for the function.

Functions declared as inline are expanded when the optimizer is invoked. Using the –x2 option automatically invokes the optimizer at the default level (–o2).

Use this syntax to define a function as inline within a module:

inline *return-type function-name* (*parameter declarations*) **{** *function* **}**

2.6.3.2 Declaring a Function as Static Inline

Declaring a function as static inline in a header file specifies that the function is inlined in any module that includes the header. This names the function and specifies to expand the function inline, but no code is generated for the function declaration itself. A function declared in this way can be placed in header files and included by all source modules of the program.

Use this syntax to declare a function as static inline:

static inline *return-type function-name* (*parameter declarations*) **{** *function* **}**

2.6.4 The _INLINE Preprocessor Symbol

The _INLINE preprocessor symbol is defined (and set to 1) if you invoke the parser (or compiler shell utility) with the –x2 (or –x) option. It allows you to write code so that it runs whether or not the optimizer is used. It is used by standard header files included with the compiler to control the declaration of standard C run-time functions.

Example 2–1 on page 2-32 illustrates how the run-time-support library uses the _INLINE preprocessor symbol.

The _INLINE preprocessor symbol is used in the string.h header file to declare the function correctly, regardless of whether inlining is used. The _INLINE preprocessor symbol conditionally defines _ _INLINE so that strlen is declared as static inline only if the _INLINE preprocessor symbol is defined.

If the rest of the modules are compiled with inlining enabled and the string.h header is included, all references to strlen are inlined and the linker does not have to use the strlen in the run-time-support library to resolve any references. Otherwise, the run-time-support library code resolves the references to strlen, and function calls are generated.

Use the _INLINE preprocessor symbol in your header files in the same way that the function libraries use it so that your programs run, regardless of whether inlining is selected for any or all of the modules in your program.

Functions declared as inline are expanded whenever the optimizer is invoked at any level. Functions declared as inline and controlled by the _INLINE preprocessor symbol, such as the run-time-library functions, are expanded whenever the optimizer is invoked and the _INLINE preprocessor symbol is equal to 1. When you declare an inline function in a library, it is recommended that you use the _INLINE preprocessor symbol to control its declaration. If you fail to control the expansion using _INLINE and subsequently compile *without* the optimizer, the call to the function is unresolved.

In Example 2–1, there are two definitions of the strlen function. The first, in the header file, is an inline definition. Note that this definition is enabled and the prototype is declared as static inline only if _INLINE is true; that is, the module including this header is compiled with the –x option.

The second definition, for the library, ensures that the callable version of strlen exists when inlining is disabled. Since this is not an inline function, the _INLINE preprocessor symbol is undefined (#undef) before string.h is included to generate a noninline version of strlen's prototype.

Example 2–1. How the Run-Time-Support Library Uses the _INLINE Preprocessor Symbol

(a) string.h

```
/****************************************************************************/
/* STRING.H HEADER FILE                                                     */
/****************************************************************************/
typdef unsigned size_t

#if _INLINE
#define __INLINE static inline    /* Declaration when inlining         */
#else
#define __INLINE                  /*No declaration when not inlining   */
#endif

__INLINE  void *memcpy (void *_s1, const void *_s2, size_t _n);

#if _INLINE                       /* Declare the inlined function      */

static inline void *memcpy (void *to, const void *from, size_t n)
{
   register char *rto   = (char *) to;
   register char *rfrom = (char *) from;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *rto++ =rfrom++;
   return (to);
}

#endif    /* _INLINE   */

#undef __INLINE
```

(b) strlen.c

```
/****************************************************************************/
/* MEMCPY.C   (rts2xx.lib)                                                  */
/****************************************************************************/
#undef _INLINE       /* Turn off so code will be generated                  */

#include <string.h>

void *memcpy (void *to, const void *from, size_t n)
{
   register char *rto   = (char *) to;
   register char *rfrom = (char *) from;
   register size_t rn;

   for (rn = 0; rn < n; rn++) *rto++ =rfrom++;
   return (to);
}
```

2.7 Using the Interlist Utility

The compiler tools include a utility that interlists C source statements into the assembly language output of the compiler. The interlist utility enables you to inspect the assembly code generated for each C statement. The interlist utility behaves differently depending on whether or not the optimizer is used, and depending on which options you specify.

The easiest way to invoke the interlist utility is to use the –s option. To compile and run the interlist utility on a program called function.c, enter:

```
dspcl -s function
```

The –s option prevents the shell from deleting the interlisted assembly language file. The output assembly file, function.asm, is assembled normally.

When you invoke the interlist utility without the optimizer, the interlist utility runs as a separate pass between the code generator and the assembler. It reads both the assembly and C source files, merges them, and writes the C statements into the assembly file as comments.

Example 2–2 shows a typical interlisted assembly file.

Example 2–2. An Interlisted Assembly Language File

```
;>>>>     main()
;>>>>        int i, j;
*******************************************
* FUNCTION DEF : _main
*******************************************
_main:
       SAR       AR0,*+
       SAR       AR1,*
       LARK      AR0,3
       LAR       AR0,*0+,AR2
;>>>>        i += j;
       LARK      AR2,1
       MAR       *0+
       LAC       *-
       ADD       *
       SACL      *+
;>>>>        j = i + 123;
       ADDK      123
       SACL      *,AR1
EPI0_1:
       SBRK      4
       LAR       AR0,*
       RET
       .end
```

For information on how to invoke the interlist utility, outside of dspcl, refer to section 2.9.5, *Invoking the Interlist Utility*, page 2-45.

Note: Using the –s Option With the Optimizer

Optimization makes normal source interlisting impractical because the optimizer extensively rearranges your program. When you use the –s option, the optimizer writes reconstructed C statements. The comments also include a list of the allocated register variables. Occasionally the optimizer interlist comments may be misleading because of copy propagation or assignment of multiple or equivalent variables to the same register.

2.8 Understanding and Handling Compiler Errors

One of the compiler's primary functions is to detect and report errors in the source program. When the compiler encounters an error in your program, it displays a message in the following format:

"*file.c***", line** *n***: [***ECODE***]** *error message*

"*file.c***"**	identifies the filename.
line *n*	identifies the line number where the error occurs.
ECODE	is a 4-character error code. A single upper-case letter identifies the error class; a 3-digit number uniquely identifies the error.
error message	is the text of the message.

Errors in C code are divided into classes according to severity; these classes are identified by the letters *W*, *E*, *F*, and *I (upper-case i)*. The compiler also reports other errors that are not related to C but prevent compilation. Examples of each level of error message are located in Table 2–3.

- **Code-W errors** are warnings resulting from a condition that is technically undefined according to the rules of the language or that can cause unexpected results. The compiler continues running when this occurs.
- **Code-E errors** are recoverable, resulting from a condition that violates the semantic rules of the language. Although these are normally fatal errors, the compiler can recover and generate an output file if you use the –pe option. See to section 2.8.2, *Treating Code-E Errors as Warnings (–pe Option)*, on page 2-36 for more information.
- **Code-F errors** are fatal, resulting from a condition that violates the syntactic or semantic rules of the language. The compiler cannot recover and does not generate output for code-F errors.
- **Code-I errors** are implementation errors, occurring when one of the compiler's internal limits is exceeded. These errors are usually caused by extreme behavior in the source code rather than by explicit errors. In most cases, code-I errors cause the compiler to abort immediately. Most code-I messages contain the maximum value for the limit that was exceeded. (Those limits that are absolute are also listed in section 5.9, *Compiler Limits*, on page 5-16.)
- **Other error messages**, such as incorrect command line syntax or inability to find specified files, are usually fatal. They are identified by the symbol >> preceding the message.

Table 2–3. Example Error Messages

Error Level	Example Error Message
Code W	`"file.c", line 42:[W029] extra text after preprocessor directive ignored`
Code E	`"file.c", line 66: [E055] illegal storage class for function 'f'`
Code F	`"file.c", line 71: [F0108] structure member 'a' undefined`
Code I	`"file.c", line 99: [I011] block nesting too deep (max=20)`
Other	`>> Cannot open source file 'mystery.c'`

2.8.1 Generating an Error Listing (–pr Option)

Use the –pr option to generate an error listing. The error listing has the name *source*.**err**, where *source* is the name of the C source file.

2.8.2 Treating Code-E Errors as Warnings (–pe Option)

A *fatal error* prevents the compiler from generating an output file. Normally, code-E, -F, and -I errors are fatal, while -W errors are not. The –pe option causes the compiler to treat code-E errors as warnings, so that the compiler generates code for the file despite the error.

Using –pe allows you to bend the rules of the language, so be careful; as with any warning, the compiler might not generate what you expect.

There is no way to specify recovery from code-F or -I errors. These errors are always fatal.

See section 2.8.4, *An Example of How You Can Use Error Options*, for an example of the –pe option.

2.8.3 Altering the Level of Warning Messages (–pw Option)

You can determine which levels of warning messages to display by setting the warning message level with the –pw option. The number following –pw denotes the level (0,1, or 2). Use Table 2–4 to select the appropriate level. See section 2.8.4, *An Example of How You Can Use Error Options*, for an example of the –pw option.

Table 2–4. Selecting a Level for the –pw Option

If you want to…	**Use option**
Disable all warning messages. This level is useful when you are aware of the condition causing the warning and consider it innocuous.	–pw0
Enable serious warning messages. This is the default.	–pw1
Enable all warning messages.	–pw2

2.8.4 An Example of How You Can Use Error Options

The following example demonstrates how you can suppress errors with the –pe option and/or alter the level of error messages with the –pw option. The examples use this code segment:

```
   int *pi; char *pc;

#if STDC
   pi = (int *) pc;
#else
   pi = pc;
#endif
```

- If you invoke the compiler with the –q option, this is the result:

  ```
  [err]
  "err.c", line3: [E104] operands of '=' point to different types
  ```

  In this case, because code-E errors are fatal, the compiler does not generate code.

- If you invoke the compiler with the –pe option, this is the result:

  ```
  [err]
  "err.c", line3: [E104] operands of '=' point to different types
  ```

  In this case, the same message is generated, but because –pe is used, the compiler ignores the error and generates an output file.

- If you invoke the compiler with the –pew2 option (combining –pe and –pw2), this is the result:

  ```
  [err.c]
  "err.c", line5: [W038] undefined preprocessor symbol 'STDC'
  "err.c", line8: [E122] operands of '=' point to different types
  ```

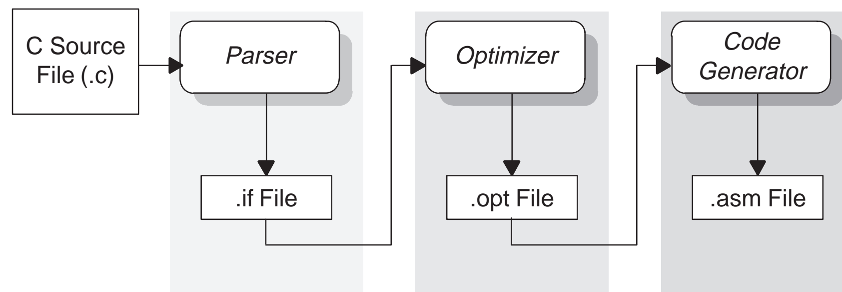
  As in the previous case, –pe causes the compiler to overlook the error and generate code. Because the –pw2 option is used, all warning messages are generated.

2.9 Invoking the Tools Individually

The TMS320C2x/C2xx/C5x C compiler offers you the versatility of invoking all of the tools at once using dspcl, or invoking each tool individually. To satisfy a variety of applications, you can invoke the compiler (parser, optimizer, and code generator), the assembler, and the linker as individual programs. This section also describes how to invoke the interlist utility outside dspcl.

- The **compiler** is composed of three distinct programs: the parser, the optimizer, and the code generator.

Figure 2–2. Compiler Overview



The input for the **parser** is a C source file. The parser reads the source file, checks for syntax and semantic errors, and writes out an internal representation of the program called an intermediate file. Section 2.9.1, *Invoking the Parser*, on page 2-39 describes how to run the parser. The parser, in addition, can be run in two passes: the first pass preprocesses the code, and the second pass parses the code.

The **optimizer** is an optional pass that runs between the parser and the code generator. The input is the intermediate file (.if) produced by the parser. When you run the optimizer, you choose the level of optimization. The optimizer performs the optimizations on the intermediate file and produces a highly efficient version of the file in the same intermediate file format. Chapter 3, *Optimizing Your Code*, describes the optimizer.

The input for the **code generator** is the intermediate file produced by the parser (.if) or the optimizer (.opt). The code generator produces an assembly language source file. Section 2.9.4, *Invoking the Code Generator,* on page 2-43, describes how to run the code generator.

- The input for the **assembler** is the assembly language file produced by the code generator. The assembler produces a COFF object file. The assembler is described fully in the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide.*

- The inputs for the **interlist utilit**y are the assembly file produced by the compiler and the C source file. The utility produces an expanded assembly source file containing statements from the C file as assembly language comments. section 2.7, *Using the Interlist Utility*, on page 2-33 and section 2.9.5, *Invoking the Interlist Utility,* on page 2-45 describe the use of the interlist utility.

- The input for the **linker** is the COFF object file produced by the assembler. The linker produces an executable object file. Chapter 4, *Linking C Code,* describes how to run the linker. The linker is described fully in the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide.*

2.9.1 Invoking the Parser

The first step in compiling a TMS320C2x/C2xx/C5x C program is to invoke the C parser. The parser reads the source file, performs preprocessing functions, checks syntax, and produces an intermediate file that can be used as input for the code generator. To invoke the parser, enter the following:

dspac *input file* [*output file*] [*options*]

dspac	is the command that invokes the parser.
input file	names the C source file that the parser uses as input. If you do not supply an extension, the parser assumes that the file's extension is *.c*. If you do not specify an input filename, the parser prompts you for one.
output file	names the intermediate file that the parser creates. If you do not supply a filename for the output file, the parser uses the input filename with the extension of *.if*.
options	affect the operation of the parser. Each option for the standalone parser has a corresponding dspcl option that performs the same function. Table 2–5 on page 2-40 shows the parser options, the dspcl shell options, and the corresponding functions.

Table 2–5. Parser Options and dspcl Options

dspac Options	dspcl Options	Effect
–d*name* [def]	–d*name* [def]	Predefine macro *name*
–e	–pe	Treat code-E errors as warnings
–f	–pf	Generate function prototype listing file
–i *dir*	–i *dir*	Define #include search path
–k	–pk	Allow K&R compatibility
–l (lowercase L)	–pl	Generate .pp file
–n	–pn	Suppress #line directives
–o	–po	Preprocess only
–q	–q	Suppress progress messages (quiet)
–tf	–tf	Relax prototype checking
–tp	–tp	Relax pointer combination
–u*name*	–u*name*	Undefine macro *name*
–v25	–v25	Enable use of TMS320C2x instructions
–v2xx	–v2xx	Enable use of TMS320C2xx instructions
–v50	–v50	Enable use of TMS320C5x instructions
–w	–pw	Suppress warning messages
–x	–x2	Enable inlining of user functions (implies –o2)
–x0	–x0	Disable function inlining
–?	–p?	Enable trigraph expansion

When running dspac stand-alone and using –l to generate a preprocessed listing file, you can specify the name of the file as the third filename on the command line. This filename can appear anywhere on the command line after the names of the source file and intermediate file.

2.9.2 Parsing in Two Passes

Compiling very large source programs on small host systems such as PCs can cause the compiler to run out of memory and fail. You can avoid host memory limitations by running the parser as two separate passes. The first pass can preprocess the file, and the second pass can parse the file.

When you run the parser as one pass, it uses host memory to store both macro definitions and symbol definitions simultaneously. When you run the parser as two passes, these functions can be separated. The first pass performs only preprocessing and memory is needed only for macro definitions. In the second pass, there are no macro definitions and memory is needed only for the symbol table.

The following example illustrates how to run the parser as two passes:

1) Run the parser with the –po option, specifying preprocessing only.

   ```
   dspcl –po file.c
   ```

   If you want to use the –d, –u, or –i options, use them on this first pass. This pass produces a preprocessed output file called file.pp. For more information about the preprocessor, see section 2.5, *Controlling the Preprocessor,* on page 2-22.

2) Rerun the whole compiler on the preprocessed file to finish compiling it.

   ```
   dspcl file.pp
   ```

   You can use any other options on this final pass.

2.9.3 Invoking the Optimizer

Optimizing is an optional second step in compiling a TMS320C2x/C2xx/C5x C program. After parsing a C source file, you can choose to process the intermediate file with the optimizer. The optimizer improves the execution speed and reduces the size of C programs. The optimizer reads the intermediate file, optimizes it according to the level you choose, and produces an intermediate file. The intermediate file has the same format as the original intermediate file, but it enables the code generator to produce more efficient code.

To invoke the optimizer, enter:

dspopt [*input file* [*output file*]] [*options*]

dspopt is the command that invokes the optimizer.

input file names the intermediate file produced by the parser. The optimizer assumes that the extension is *.if*. If you do not specify an input file, the optimizer prompts you for one.

output file names the intermediate file that the optimizer creates. If you do not supply a filename for the output file, the optimizer uses the input filename with the extension *.opt*.

options affect how the the optimizer processes the input file. The options that you use in stand-alone optimization are the same as those used for dspcl. Table 2–6 shows the optimizer options, the dspcl shell options, and the corresponding functions. Section 3.1, *Using the C Compiler Optimizer*, on page 3-2 provides a detailed list of the optimizations performed at each level.

Table 2–6. Optimizer Options and dspcl Options

dspopt Option	dspcl Option	Function
–a	–ma	Assume variables are aliased
–b	–mb	Disable the noninterruptible RPTK instruction for moving structures
–g*register*	–r*register*	Reserve global register†
–h*size*	–ol*size*	Control assumptions about library function calls
–i*n*	–oi*n*	Set automatic inlining size threshold (–o3 only)
–j	–oe	Assume no functions call, or are called by, interrupt functions
–k	–pk	Allow K&R compatibility
–n*n*	–on*n*	Generate optimization information file (–o3 only)
–o0	–o0	Optimize at level 0; register optimization
–o1	–o1	Optimize at level 1; + local optimization

† The **–g** option tells the code generator that the register named is reserved for global use. See section 5.6, *Creating Global Register Variables*, on page 5-10, for more information.

Table 2–6. Optimizer Options and dspcl Options (Continued)

dspopt Option	dspcl Option	Function
–o2 or –o	–o2	Optimize at level 2; + global optimization
–o3	–o3	Optimize at level 3; + file optimization
–q	–q	Suppress progress messages (quiet)
–s	–s	Interlist C source
–v25	–v25	Enable use of TMS320C2x instructions
–v2xx	–v2xx	Enable use of TMS320C2xx instructions
–v50	–v50	Enable use of TMS320C5x instructions

† The **–g** option tells the code generator that the register named is reserved for global use. See section 5.6, *Creating Global Register Variables*, on page 5-10, for more information.

2.9.4 Invoking the Code Generator

The third step in compiling a TMS320C2x/C2xx/C5x C program is to invoke the C code generator. The code generator converts the intermediate file produced by the parser into an assembly language source file. You can modify this output file or use it as input for the assembler. The code generator produces re-entrant relocatable code, which, after assembling and linking, can be stored in ROM.

To invoke the code generator as a stand-alone program, enter:

dspcg [*input file* [*output file* [*tempfile*]]] [*options*]

dspcg is the command that invokes the code generator.

input file names the intermediate file that the code generator uses as input. If you do not supply an extension, the code generator assumes that the extension is *.if*. If you do not specify an input file, the code generator prompts you for one.

output file names the assembly language file that the code generator creates. If you do not supply a filename for the output file, the code generator uses the input filename with the extension *.asm*.

tempfile names a temporary file that the code generator creates and uses. If you do not supply a filename for the temporary file, the code generator uses the input filename with the extension *.tmp*. The code generator deletes this file after using it.

options affect the way the code generator processes the input file. Each option available for the stand-alone code generator mode has a corresponding dspcl shell option that performs the same function.

Table 2–7 shows the code generator options, the dspcl shell options, and their corresponding functions.

Table 2–7. Code Generator Options and dspcl Options

dspcg Options	dspcl Options	Function
–a	–ma	Assume variables are aliased
–b	–mb	Disable the noninterruptible RPTK instruction for moving structures
–g*register*	–r*register*	Reserve global register†
–l	–ml	Disable optimization for reducing LDPK instructions
–n	–mn	Reenable optimizations disabled by symbolic debugging
–o	–g	Enable C source level debugging
–q	–q	Suppress progress messages (quiet)
–r	–mr	List register use information
–s	–ms	Optimize for space instead of for speed
–v25	–v25	Enable use of TMS320C2x instructions
–v2xx	–v2xx	Enable use of TMS320C2xx instructions
–v50	–v50	Enable use of TMS320C5x instructions
–x	–mx	Avoid ’C5x silicon bugs
–z		Retain the input file‡

† The **–g** option tells the code generator that the register named is reserved for global use. See section 5.6, *Creating Global Register Variables*, on page 5-10, for more information.
‡ The **–z** option tells the code generator to retain the input file (the intermediate file created by the parser). If you do not specify the –z option, the intermediate file is deleted.

2.9.5 Invoking the Interlist Utility

The fourth step in compiling a TMS320C2x/C2xx/C5x C program is optional. After you have compiled a program, you can run the interlist utility as a standalone program. To run the interlist utility from the command line, the syntax is:

clist *asmfile* [*outfile*] [*options*]

clist is the command that invokes the interlist utility.

asmfile is the assembly language output from the compiler.

outfile names the interlisted output file. If you do not supply a filename for the outfile, the interlist utility uses the assembly language filename with the extension *.cl*.

options control the operation of the utility as follows:

- **–b** removes blanks and useless lines (lines containing comments and lines containing only { or }).
- **–q** removes banner and status information.
- **–r** removes symbolic debugging directives.

The interlist utility uses .line directives produced by the code generator to associate assembly language code with C source. For this reason, you must use the –g dspcl option to specify symbolic debugging when compiling the program if you want to interlist it. If you do not want the debugging directives in the output, use the –r interlist option to remove them from the interlisted file.

The following example shows how to compile and interlist function.c. To compile, enter:

```
dspcl –gk –mn function
```

This compiles, produces symbolic debugging directives, and keeps the assembly language file. To produce an interlist file, enter:

```
clist –r function
```

This creates an interlist file and removes the symbolic debugging directives. The output from this example is function.cl.

Chapter 3

Optimizing Your Code

The compiler tools include an optimization program that improves the execution speed and reduces the size of C programs by performing such tasks as simplifying loops, software pipelining, rearranging statements and expressions, and allocating variables into registers.

This chapter describes how to invoke the optimizer and describes which optimizations are performed when you use it. This chapter also describes how you can use the interlist utility with the optimizer and how you can profile or debug optimized code.

Topic **Page**

3.1 Using the C Compiler Optimizer 3-2

3.2 Using the –o3 Option 3-4

3.3 Performing Program-Level Optimization (–pm and –o3 Options) 3-6

3.4 Special Considerations When Using the Optimizer 3-10

3.5 Automatic Inline Expansion (–oi Option) 3-12

3.6 Using the Interlist Utility With the Optimizer 3-13

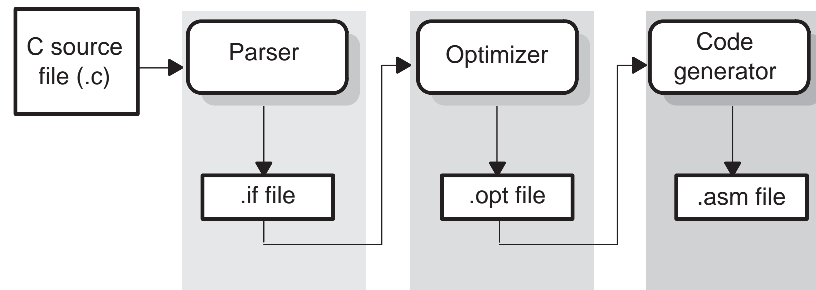
3.7 Debugging Optimized Code 3-13

3.8 What Kind of Optimization Is Being Performed? 3-14

3.1 Using the C Compiler Optimizer

The optimizer runs as a separate pass between the parser and the code generator. Figure 3–1 illustrates the execution flow of the compiler with standalone optimization.

Figure 3–1. Compiling a C Program With the Optimizer



The easiest way to invoke the optimizer is to use the dspcl shell program, specifying the –o*n* option on the dspcl command line. The *n* denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization:

- **–o0**
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates unused code
  - Simplifies expressions and statements
  - Expands calls to functions declared inline
- **–o1**

  Performs all –o0 optimizations, plus:
  - Performs local copy/constant propagation
  - Removes unused assignments
  - Eliminates local common expressions
- **–o2**

  Performs all –o1 optimizations, plus:
  - Performs loop optimizations
  - Eliminates global common subexpressions
  - Eliminates global unused assignments
  - Converts array references in loops to incremented pointer form
  - Performs loop unrolling

  The optimizer uses –o2 as the default if you use –o without an optimization level.

- **–o3**

  Performs all –o2 optimizations, plus:

  - Removes all functions that are never called
  - Simplifies functions with return values that are never used
  - Inlines calls to small functions
  - Reorders function declarations so that the attributes of called functions are known when the caller is optimized
  - Propagates arguments into function bodies when all calls pass the same value in the same argument position
  - Identifies file-level variable characteristics

  If you use –o3, see section 3.2, *Using the –o3 Option*, on page 3-4 for more information.

The levels of optimization described above are performed by the stand-alone optimization pass. The code generator performs several additional optimizations, particularly ’C6000-specific optimizations; it does so regardless of whether or not you invoke the optimizer. These optimizations are always enabled and are not affected by the optimization level you choose.

You can also invoke the optimizer outside dspcl; see section 2.9.3, *Invoking the Optimizer*, on page 2-41 for information about invoking the optimizer as a separate step.

3.2 Using the –o3 Option

The –o3 option instructs the compiler to perform file-level optimization. You can use the –o3 option alone to perform general file-level optimization, or you can combine it with other options to perform more specific optimizations. The options listed in Table 3–1 work with –o3 to perform the indicated optimization:

Table 3–1. Options That You Can Use With –o3

If you ...	Use this option	Page
Have files that redeclare standard library functions	–ol*n*	3-4
Want to create an optimization information file	–on*n*	3-5
Want to compile multiple source files	–pm	3-6

3.2.1 Controlling File-Level Optimization (–ol*n* Option)

When you invoke the optimizer with the –o3 option, some of the optimizations use known properties of the standard library functions. If your file redeclares any of these standard library functions, these optimizations become ineffective. The –ol (lowercase L) option controls file-level optimizations. The number following the –ol denotes the level (0, 1, or 2). Use Table 3–2 to select the appropriate level to append to the –ol option.

Table 3–2. Selecting a Level for the –ol Option

If your source file...	Use this option
Declares a function with the same name as a standard library function	–ol0
Contains but does not alter functions declared in the standard library	–ol1
Does not alter standard library functions, but you used the –ol0 or –ol1 option in a command file or an environment variable. The –ol2 option restores the default behavior of the optimizer.	–ol2

3.2.2 Creating an Optimization Information File (–on*n* Option)

When you invoke the optimizer with the –o3 option, you can use the –on option to create an optimization information file that you can read. The number following the –on denotes the level (0, 1, or 2). The resulting file has an .nfo extension. Use Table 3–3 to select the appropriate level to append to the –on option.

Table 3–3. Selecting a Level for the –on Option

If you...	Use this option
Do not want to produce an information file, but you used the –on1 or –on2 option in a command file or an environment variable. The –on0 option restores the default behavior of the optimizer.	–on0
Want to produce an optimization information file	–on1
Want to produce a verbose optimization information file	–on2

3.3 Performing Program-Level Optimization (–pm and –o3 Options)

You can specify program-level optimization by using the –pm option with the –o3 option. With program-level optimization, all of your source files are compiled into one intermediate file called a *module*. The module moves to the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization:

- If a particular argument in a function always has the same value, the compiler replaces the argument with the value and passes the value instead of the argument.
- If a return value of a function is never used, the compiler deletes the return code in the function.
- If a function is not called, directly or indirectly, the compiler removes the function.

To see which program-level optimizations the compiler is applying, use the –on2 option to generate an information file. See section 3.2.2, *Creating an Optimization Information File (–on*n *Option)*, on page 3-5 for more information.

3.3.1 Controlling Program-Level Optimization (–op*n* Option)

You can control program-level optimization, which you invoke with –pm –o3, by using the –op option. Specifically, the –op option indicates if functions in other modules can call a module's external functions or modify a module's external variables. The number following –op indicates the level you set for the module that you are allowing to be called or modified. The –o3 option combines this information with its own file-level analysis to decide whether to treat this module's external function and variable declarations as if they had been declared static. Use Table 3–4 to select the appropriate level to append to the –op option.

Table 3–4. Selecting a Level for the –op Option

If your module …	**Use this option**
Has functions that are called from other modules and global variables that are modified in other modules	–op0
Does not have functions that are called by other modules but has global variables that are modified in other modules	–op1
Does not have functions that are called by other modules or global variables that are modified in other modules	–op2
Has functions that are called from other modules but does not have global variables that are modified in other modules	–op3

In certain circumstances, the compiler reverts to a different –op level from the one you specified, or it might disable program-level optimization altogether. Table 3–5 lists the combinations of –op levels and conditions that cause the compiler to revert to other –op levels.

Table 3–5. Special Considerations When Using the –op Option

If your –op is...	**Under these conditions...**	**Then the –op level...**
Not specified	The –o3 optimization level was specified	Defaults to –op2
Not specified	The compiler sees calls to outside functions under the –o3 optimization level	Reverts to –op0
Not specified	Main is not defined	Reverts to –op0
–op1 or –op2	No function has main defined as an entry point	Reverts to –op0
–op1 or –op2	No interrupt function is defined	Reverts to –op0
–op1 or –op2	Functions are identified by the FUNC_EXT_CALLED pragma	Reverts to –op0
–op3	Any condition	Remains –op3

In some situations when you use –pm and –o3, you *must* use an –op option or the FUNC_EXT_CALLED pragma. See section 3.3.2, *Optimization Considerations When Mixing C and Assembly*, on page 3-8 for information about these situations.

3.3.2 Optimization Considerations When Mixing C and Assembly

If you have any assembly functions in your program, you need to exercise caution when using the –pm option. The compiler recognizes only the C source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C functions, the –pm option optimizes out those C functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see section 5.4.3, *The FUNC_EXT_CALLED Pragma*, on page 5-8) before any declaration or reference to a function that you want to keep.

Another approach you can take when you use assembly functions in your program is to use the –op*n* option with the –pm and –o3 options (see section 3.3.1, *Controlling Program-Level Optimization*, on page 3-6).

In general, you achieve the best results through judicious use of the FUNC_EXT_CALLED pragma in combination with –pm –o3 and –op1 or –op2.

If any of the following situations apply to your application, use the suggested solution:

Situation Your application consists of C source code that calls assembly functions. Those assembly functions do not call any C functions or modify any C variables.

Solution Compile with –pm –o3 –op2 to tell the compiler that outside functions do not call C functions or modify C variables.

If you compile with the –pm –o3 options only, the compiler reverts from the default optimization level (–op2) to –op0. The compiler uses –op0, because it presumes that the calls to the assembly language functions that have a definition in C may call other C functions or modify C variables.

Situation Your application consists of C source code that calls assembly functions. The assembly language functions do not call C functions, but they modify C variables.

Solution Try both of these solutions and choose the one that works best with your code:

- Compile with –pm –o3 –op1.
- Add the volatile keyword to those variables that may be modified by the assembly functions and compile with –pm –o3 –op2.

See section 3.3.1 on page 3-6 for information about the –op*n* option.

Situation Your application consists of C source code and assembly source code. The assembly functions are interrupt service routines that call C functions; the C functions that the assembly functions call are never called from C. These C functions act like main: they function as entry points into C.

Solution Add the volatile keyword to the C variables that may be modified by the interrupts. Then, you can optimize your code in one of these ways:

- You achieve the best optimization by applying the FUNC_EXT_CALLED pragma to all of the entry-point functions called from the assembly language interrupts, and then compiling with –pm –o3 –op2. *Be sure that you use the pragma with all of the entry-point functions.* If you do not, the compiler removes the entry-point functions that are not preceded by the FUNC_EXT_CALL pragma.
- Compile with –pm –o3 –op3. Because you do not use the FUNC_EXT_CALL pragma, you must use the –op3 option, which is less aggressive than the –op2 option, and your optimization may not be as effective.

Keep in mind that if you use –pm –o3 without additional options, the compiler removes the C functions that the assembly functions call. Use the FUNC_EXT_CALLED pragma to keep these functions.

Use the –on2 option to generate an information file to see which program-level optimizations the compiler is applying. See section 3.2.2, *Creating an Optimizaton Information File*, on page 3-5 for more information.

3.3.3 Naming the Program Compilation Output File (–px Option)

When you specify whole program compilation with the –pm option, you can use the –px *filename* option to specify the name of the output file. If you specify no assembly (–n shell option), the default file extension for the output file is .asm. If you allow assembly (default shell behavior), the default file extension for the output file is .obj. If you specify linking, you must name the output file with the –o option after the –z option, or the name of the output file is the default a.out.

3.4 Special Considerations When Using the Optimizer

The optimizer is designed to improve your ANSI-conforming C programs while maintaining their correctness. However, when you write code for the optimizer, you should note the following special considerations to ensure that your program performs as you intend.

3.4.1 Use Caution With asm Statements in Optimized Code

You must be extremely careful when using asm (inline assembly) statements in optimized code. The optimizer rearranges code segments, uses registers freely, and can completely remove variables or expressions. Although the compiler never optimizes out an asm statement (except when it is unreachable), the surrounding environment where the assembly code is inserted can differ significantly from the original C source code. It is usually safe to use asm statements to manipulate hardware controls such as interrupt masks, but asm statements that attempt to interface with the C environment or access C variables can have unexpected results. After compilation, check the assembly output to make sure your asm statements are correct and maintain the integrity of the program.

3.4.2 Use Caution With the Volatile Keyword

The optimizer analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the C code, you must use the volatile keyword to identify these accesses. A variable qualified with a volatile keyword is allocated to an uninitialized section. The compiler will not optimize out any references to volatile variables.

In the following example, the loop waits for a location to be read as 0xFF:

```
unsigned int *ctrl;
while (*ctrl !=0xFF);
```

In this example, `*ctrl` is a loop-invariant expression, so the loop is optimized down to a single memory read. To correct this, declare `ctrl` as:

```
volatile unsigned int *ctrl
```

3.4.3 Use Caution When Accessing Aliased Variables

Aliasing occurs when you can access a single object in more than one way, such as when two pointers point to the same object or when a pointer points to a named object. Aliasing can disrupt optimization, because any indirect reference could potentially refer to any other object. The optimizer analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while still preserving the correctness of the program. The optimizer behaves conservatively. If there is a chance that two pointers are pointing to the same object, the optimizer assumes that the pointers point to the same object.

The compiler assumes that if the address of a local variable is passed to a function, the function might change the local variable by writing through the pointer. This makes its address unavailable for use elsewhere after returning. For example, the called function cannot assign the local variable's address to a global variable or return the local variable's address.

3.4.4 Assume Functions Are Not Interrupts

The –oe option assumes that none of the functions in the module are interrupts, can be called by interrupts, or can be otherwise executed in an asynchronous manner. This enables the optimizer to do certain variable allocation optimizations. The –oe option automatically invokes the optimizer at level 2.

The –oe option also presumes that none of the modules are called recursively (directly or indirectly). Be careful not to combine the use of –oe with modules containing recursive functions.

3.5 Automatic Inline Expansion (–oi Option)

The optimizer automatically inlines small functions when it is invoked with the –o3 option. A command-line option, –oi*size*, specifies the size of the functions inlined. When you use –oi, specify the *size* limit for the largest function to be inlined. You can use the –oi*size* option in the following ways:

- If you set the *size* parameter to 0 (–oi0), all size-controlled inlining is disabled.
- If you set the *size* parameter to a nonzero integer, the compiler inlines functions based on *size*. The optimizer multiplies the number of times the function is inlined (plus 1 if the function is externally visible and its declaration cannot be safely removed) by the size of the function. The optimizer inlines the function only if the result is less than the size parameter. The compiler measures the size of a function in arbitrary units; however, the optimizer information file (created with the –on1 or –on2 option) reports the size of each function in the same units that the –oi option uses.

The –oi*size* option controls only the inlining of functions that are not explicitly declared as inline. If you do not use the –oi*size* option, the optimizer inlines very small functions. The –x option controls the inlining of functions declared as inline (see section 2.6.3.1 on page 2-30).

3.6 Using the Interlist Utility With the Optimizer

You control the output of the interlist utility when running the optimizer (the –o*n* option) with the –os and –ss options.

- The –os option interlists optimizer comments with assembly source statements.
- The –ss and –os options together interlist the optimizer comments and the original C source with the assembly code.

When you use the –os option with the optimizer, the interlist utility does *not* run as a separate pass. Instead, the optimizer inserts comments into the code, indicating how the optimizer has rearranged and optimized the code. These comments appear in the assembly language file as comments starting with ;**. The C source code is not interlisted, unless you use the –ss option also.

The interlist utility can affect optimized code because it might prevent some optimization from crossing C statement boundaries. Optimization makes normal source interlisting impractical, because the optimizer extensively rearranges your program. Therefore, when you use the –os option, the optimizer writes reconstructed C statements.

3.7 Debugging Optimized Code

Ideally, you should debug a program in an unoptimized form and reverify its correctness after it has been optimized. You can use the debugger with optimized code, but the optimizer's extensive rearrangement of code and the many-to-one allocation of variables to registers often makes it difficult to correlate source code with object code.

Note: Symbolic Debugging and Optimized Code

If you use the –g option to generate symbolic debugging information, many code generator optimizations are disabled because they disrupt the debugger. If you want to use symbolic debugging and still generate fully optimized code, use the –mn option. –mn re-enables the optimizations disabled by –g.

3.8 What Kind of Optimization Is Being Performed?

The TMS320C2x/C2xx/C5x C compiler uses a variety of optimization techniques to improve the execution speed of your C programs and to reduce their size. Optimization occurs at various levels throughout the compiler.

Most of the optimizations described here are performed by the separate optimizer pass that you enable and control with the –o compiler options (see section 3.1, *Using the Compiler Optimizer*, on page 3-2). However, the code generator performs some optimizations, which you cannot selectively enable or disable.

Following are the optimizations performed by the compiler. These optimizations improve any C code:

Optimizations	Page
Cost-based register allocation	3-15
Autoincrement addressing	3-15
Repeat Blocks	3-15
Delays, branches, calls, and returns	3-16
Algebraic reordering, symbolic simplification, constant folding	3-18
Alias disambiguation	3-18
Data flow optimizations ❑ Copy propagation ❑ Common subexpression elimination ❑ Redundant assignment elimination	3-18
Branch optimizations and control-flow simplification	3-20
Loop induction variable optimizations and strength reduction	3-21
Loop rotation	3-21
Loop-invariant code motion	3-21
Inline expansion of run-time-support library functions	3-21

3.8.1 Cost-based Register Allocation

The optimizer, when enabled, allocates registers to user variables and compiler temporary values according to their type, use, and frequency. Variables used within loops are weighted to have priority over others, and those variables whose uses do not overlap may be allocated to the same register.

3.8.2 Autoincrement Addressing

For pointer expressions of the form *p++, the compiler uses efficient TMS320C2x/C2xx/C5x autoincrement addressing modes. In many cases, where code steps through an array in a loop, such as for (i = 0; i < n; ++i) a[i]..., the loop optimizations convert the array references to indirect references through autoincremented register variable pointers. See Example 3–1.

3.8.3 Repeat Blocks

The TMS320C2x/C2xx/C5x supports zero-overhead loops with the RPTB (repeat block) instruction. With the optimizer, the compiler can detect loops controlled by counters and generate them using the efficient repeat forms. The iteration count can be either a constant or an expression. For the TMS320C2x, which does not have a repeat block instruction, the compiler allocates an AR as the loop counter and implements the loop with a BANZ instruction. See Example 3–1 and Example 3–5.

Example 3–1. Repeat Blocks, Autoincrement Addressing, Parallel Instructions, Strength Reduction, Induction Variable Elimination, Register Variables, and Loop Test Replacement

```
int a[10], b[10];
scale(int k)
{
    int i;
    for (i = 0; i < 10; ++i)
      a[i] = b[i] * k;
   . . .
```

TMS320C2x/C2xx/C5x C Compiler Output:

```
_scale:
   . . .
   LRLK   AR6,_a           ; AR6 = &a[0]
   LRLK   AR5,_b           ; AR5 = &b[0]
   LACK   9
   SAMM   BRCR             ; BRCR = 9
   LARK   AR2,-3+LF1       ; AR2 = &k
   MAR    *0+,AR5
   RPTB   L4-1             ; repeat block 10 times
   LT     *+,AR2           ; t = *AR5++
   MPY    * ,AR6           ; p = t * *AR2
   SPL    *+,AR5           ; *AR6++ = p
L4:
   . . .
```

Induction variable elimination and loop test replacement allow the compiler to recognize the loop as a simple counting loop and then generate a repeat block. Strength reduction turns the array references into efficient pointer autoincrements.

3.8.4 Delays, Banches, Calls, and Returns

The TMS320C5x provides a number of of delayed branch, call, and return instructions. Three of these are used by the compiler: branch unconditional (BD), call to a named function (CALLD), and simple return (RETD). These instructions execute in two fewer cycles than their nondelayed counterparts. They execute two instructions words after they enter the instruction stream. Sometimes it is necessary to insert a NOP after a delayed instruction to ensure proper operation of the sequence. This is one word of code longer than a nondelayed sequence, but it is still one cycle faster. Note that the compiler emits a comment in the instruction sequence where the delayed instruction executes. See Example 3–2.

Example 3–2. Delayed Branch, Call, and Return Instructions

```
main()
{
    int i0, i1;

    while (input(&i0) && input(&i1))
      process(i0, i1);
}
```

TMS320C2x/C2xx/C5x C Compiler Output:

```
_main:
      SAR    AR0,*+             ; function prolog
      POPD   *+                 ; save AR0 and return address
      SAR    AR1,*              ; begin to set up local frame
      BD     L2                 ; begin branch to loop control
      LARK   AR0,3              ; finish setting up local frame
      LAR    AR0,*0+

***   B      L2 OCCURS          ; branch to loop control
L1:                             ; loop body
      LARK   AR2,2              ; AR2 = &i1
      MAR    *0+
      LAC    *-,AR1             ; ACC = *AR2, AR2 = &i0
      SACL   *+,AR2             ; stack ACC
      CALLD  _process           ; begin call
      LAC    * ,AR1             ; ACC = *AR2
      SACL   *+                 ; stack ACC
***   CALL   _process OCCURS    ; call occurs
      SBRK   2                  ; pop stack
L2:                             ; loop control
      MAR    * ,AR5             ; AR5 = &i0
      LARK   AR5,1
      CALLD  _input             ; begin call
      MAR    *0+,AR1
      SAR    AR5,*+             ; stack AR5
***   CALL   _input OCCURS      ; call occurs
      MAR    *-                 ; clear stack
      BZ     EPI0_1             ; quit if _input returns 0
      MAR    * ,AR4             ; AR4 = &i1
      LARK   AR4,2
      CALLD  _input             ; begin call
      MAR    *0+,AR1
      SAR    AR4,*+             ; stack AR4
***   CALL   _input OCCURS      ; call occurs
      MAR    *-,AR2             ; clear stack
      BNZ    L1                 ; continue if _input returns !0
EPI0_1:
      MAR    * ,AR1             ; function epilog
      SBRK   4                  ; clear local frame
      PSHD   *-                 ; push return address on hardware stack
      RETD                      ; begin return
      LAR    AR0,*              ; restore AR0
      NOP                       ; necessary, no PSHD in delay slot
***   RET OCCURS                ; return occurs
      . . .
```

3.8.5 Algebraic Reordering / Symbolic Simplification / Constant Folding

For optimal evaluation, the compiler simplifies expressions into equivalent forms requiring fewer instructions or registers. For example, the expression (a + b) – (c + d) takes six instructions to evaluate; it can be optimized to ((a + b) – c) – d, which takes only four instructions. Operations between constants are folded into single constants. For example, a = (b + 4) – (c + 1) becomes a = b – c + 3. See Example 3–3.

3.8.6 Alias Disambiguation

Programs written in the C language generally use many pointer variables. Frequently, compilers are unable to determine whether or not two or more **l** (lowercase L) values (symbols, pointer references, or structure references) refer to the same memory location. This *aliasing* of memory locations often prevents the compiler from retaining values in registers because it cannot be sure that the register and memory continue to hold the same values over time. Alias disambiguation is a technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

3.8.7 Data-Flow Optimizations

Collectively, the following three data-flow optimizations replace expressions with less costly ones, detect and remove unnecessary assignments, and avoid operations that produce values already computed. The optimizer performs these data-flow optimizations both locally (within basic blocks) and globally (across entire functions). See Example 3–3 and Example 3–4.

- **Copy propagation**

  Following an assignment to a variable, the compiler replaces references to the variable with its value. The value could be another variable, a constant, or a common subexpression. This may result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable. See Example 3–3 and Example 3–4.

- **Common subexpression elimination**

  When the same value is produced by two or more expressions, the compiler computes the value once, saves it, and reuses it. See Example 3–3.

- **Redundant assignment elimination**

  Often, copy propagation and common subexpression elimination optimizations result in unnecessary assignments to variables (variables with no subsequent reference before another assignment or before the end of the function). The optimizer removes these dead assignments. See Example 3–3.

Example 3–3. Data-Flow Optimizations

```
simp(int j)
{
    int a = 3;
    int b = (j * a) + (j * 2);
    int c = (j << a);
    int d = (j >> 3) + (j << b);

    call(a,b,c,d);
    ...
}
```

TMS320C2x/C2xx/C5x C Compiler Output:

```
_simp:
    . . .
****************************************************
* b = j * 5;
****************************************************
      LARK   AR2,-3+LF1       ; AR2 = &j
      MAR    *0+
      LT     *                ; t = *AR2
      MPYK   5                ; p = t * 5
      ADRK   4-LF1            ; AR2 = &b
      SPL    *                ; *AR2 = p
****************************************************
* call(3, b, j << 3, (j >> 3) + (j << b));
****************************************************
      LT     *                ; t = *AR2 (b)
      SBRK   4-LF1            ; AR2 = &j
      LACT   * ,AR1           ; ACC = j << b
      SACL   * ,AR2           ; save off ACC on TOS (top of stack)
      SSXM                    ; need sign extension for right shift
      LAC    * ,12,AR1        ; high ACC = j >> 3
      ADD    * ,15            ; add TOS to high ACC
      SACH   *+,1,AR2         ; stack high ACC
      LAC    * ,3,AR1         ; ACC = j << 3
      SACL   *+,AR2           ; stack ACC
      ADRK   4-LF1            ; AR2 = &b
      LAC    * ,AR1           ; ACC = b
      SACL   *+               ; stack ACC
      CALLD  _call            ; call begins
      LACK   3                ; ACC = 3
      SACL   *+               ; stack ACC
***   CALL   _call OCCURS     ; call occurs
      . . .
```

The constant 3, assigned to a, is copy propagated to all uses of a; a becomes a dead variable and is eliminated. The sum of multiplying j by 3 (a) and 2 is simplified into b = j * 5, which is recognized as a common subexpression. The assignments to c and d are dead and are replaced with their expressions. These optimizations are performed across jumps.

3.8.8 Branch Optimizations and Control-Flow Simplification

The compiler analyzes the branching behavior of a program and rearranges the linear sequences of operations (basic blocks) to remove branches or redundant conditions. Unreachable code is deleted, branches to branches are bypassed, and conditional branches over unconditional branches are simplified to a single conditional branch. When the value of a condition can be determined at compile time (through copy propagation or other data flow analysis), a conditional branch can be deleted. Switch case lists are analyzed in the same way as conditional branches and are sometimes eliminated entirely. Some simple control-flow constructs can be reduced to conditional instructions, totally eliminating the need for branches. See Example 3–4.

Example 3–4. Copy Propagation and Control-Flow Simplification

```
fsm()
{
  enum { ALPHA, BETA, GAMMA, OMEGA } state = ALPHA;
  int *input;

  while (state != OMEGA)
    switch (state)
    {
      case ALPHA: state = ( *input++ == 0 ) ? BETA:  GAMMA; break;
      case BETA : state = ( *input++ == 0 ) ? GAMMA: ALPHA; break;
      case GAMMA: state = ( *input++ == 0 ) ? GAMMA: OMEGA; break;
    }
}
```

TMS320C2x/C2xx/C5x C Compiler Output:

```
_fsm:
      . . .
*
* AR5 assigned to variable 'input'
*
      LAC    *+       ; initial state == ALPHA
      BNZ    L5       ; if (input != 0) go to state GAMMA
L2:
      LAC    *+       ; state == BETA
      BZ     L4       ; if (input == 0) go to state GAMMA
      LAC    *+       ; state == ALPHA
      BZ     L2       ; if (input == 0) go to state BETA
      B      L5       ; else            go to state GAMMA
L4:
      LAC    *+       ; state == GAMMA
      BNZ    EPI0_1   ; if (input != 0) go to state OMEGA
L5:
      LARP   AR5
L6:
      LAC    *+       ; state = GAMMA
      BZ     L6       ; if (input == 0) go to state GAMMA
EPI0_1:               ; state == OMEGA
      ...
```

The switch statement and the state variable from this simple finite state machine example are optimized completely away, leaving a streamlined series of conditional branches.

3.8.9 Loop Induction Variable Optimizations and Strength Reduction

Loop induction variables are variables whose value within a loop is directly related to the number of executions of the loop. Array indices and control variables for loops are very often induction variables. Strength reduction is the process of replacing costly expressions involving induction variables with more efficient expressions. For example, code that indexes into a sequence of array elements is replaced with code that increments a pointer through the array. Loops controlled by incrementing a counter are written as TMS320C2x/C2xx/C5x repeat blocks or by using efficient decrement-and-branch instructions. Induction variable analysis and strength reduction together often remove all references to your loop control variable, allowing it to be eliminated entirely. See Example 3–1 and Example 3–5.

3.8.10 Loop Rotation

The compiler evaluates loop conditionals at the bottom of loops, saving a costly extra branch out of the loop. In many cases, the initial entry conditional check and the branch are optimized out.

3.8.11 Loop Invariant Code Motion

This optimization identifies expressions within loops that always compute the same value. The computation is moved in front of the loop, and each occurrence of the expression in the loop is replaced by a reference to the precomputed value. See Example 3–5.

3.8.12 Inline Expansion of Run–Time-Support Library Functions

The compiler replaces calls to small run-time-support functions with inline code, saving the overhead associated with a function call, as well as providing increased opportunities to apply other optimizations. See Example 3–5.

Example 3–5. Inline Function Expansion

```
#include <string.h>
struct s { int a,b,c[10]; };
struct t { int x,y,z[10]; };

proc_str(struct s *ps, struct t *pt)
{
    . . .
    memcpy(ps,pt,sizeof(*ps));
    . . .
}
_proc_str:
    . . .
```

TMS320C2x/C2xx/C5x C Compiler Output:

```
*
* AR5 assigned to variable 'memcpy_1_rfrom'
* AR6 assigned to variable 'memcpy_1_rto'
* BRCR      assigned to temp var 'L$1'
*
      . . .

      LARK   AR2,-3+LF1 ; AR2 = &ps
      MAR    *0+
      LAR    AR6,*-     ; AR6 = ps, AR2 = &pt
      LAR    AR5,* ,AR5 ; AR5 = pt
      LACK   11
      SAMM   BRCR       ; repeat 12 times
      RPTB   L4-1
      LAC    *+,AR6     ; *ps++ = *pt++
      SACL   *+,AR5
      NOP               ; must have 3 words in repeat block
L4:
      . . .
```

The compiler finds the intermediate file code for the C function memcpy( ) in the inline library and copies it in place of the call. Note the creation of variables memcpy_1_from and memcpy_1_to, corresponding to the parameters of memcpy. (Often, copy propagation can eliminate such assignments to parameters of inlined functions when the arguments are not referenced after the call.)

Chapter 4

Linking C Code

The C compiler and assembly language tools provide two methods for linking your programs:

- You can compile individual modules and link them together. This method is especially useful when you have multiple source files.
- You can compile and link in one step by using dspcl. This method is useful when you have a single source module.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C code, including the run-time-support libraries, specifying the type of initialization, and allocating the program into memory. For a complete description of the linker, see the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

Topic **Page**

4.1 Invoking the Linker as an Individual Program 4-2
4.2 Invoking the Linker With the Compiler Shell (–z Option) 4-4
4.3 Disabling the Linker (–c Shell Option) 4-5
4.4 Linker Options 4-6
4.5 Controlling the Linking Process 4-8

4.1 Invoking the Linker as an Individual Program

This section shows how to invoke the linker in a separate step after you have compiled and assembled your programs. This is the general syntax for linking C programs in a separate step:

dsplnk {**–c**|**–cr**} *filenames* [*-options*] [**–o** *name.out*] **–l** *libraryname* [*lnk.cmd*]

lnk6x is the command that invokes the linker.

–c | **–cr** are options that tell the linker to use special conventions defined by the C environment. When you use dsplnk, you must use –c or –cr. The –c option uses automatic variable initialization at runtime; the –cr option uses variable initialization at load time.

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

options affect how the linker handles your object files. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 4.4)

–o *name.out* The –o option names the output file.

–l *libraryname* (lowercase L) Identifies the appropriate archive library containing C run-time-support and floating-point math functions. (The –l option tells the linker that a file is an archive library.) If you are linking C code, you must use a run-time-support library. You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a linker command file, you do not need this parameter.

lnk.cmd contains options, filenames, directives, or commands for the linker.

Table 4–1. Run-Time-Support Source Libraries

Library Name	Library Source Contents
rts25.lib	TMS320C2x runtime support
rts2xx.lib	TMS320C2xx runtime support
rts50.lib	TMS320C5x runtime support

When you specify a library as linker input, the linker includes and links only those library members that resolve undefined references. For example, you can link a C program consisting of modules prog1, prog2, and prog3 (the output file is named prog.out), enter:

```
dsplnk –c prog1 prog2 prog3 –o prog.out –l rts25.lib
```

The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the linker command file to customize the allocation process. For more information, see the *TMS320C2x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

4.2 Invoking the Linker With the Compiler Shell (–z Option)

The options and parameters discussed in this section apply to both methods of linking; however, when you link while compiling, the linker options must follow the –z option (see section 2.2, *Invoking the C Compiler Shell,* on page 2-4).

By default, the compiler does not run the linker. However, if you use the –z option, a program is compiled, assembled, and linked in one step. When using –z to enable linking, remember that:

- The –z option divides the command line into compiler options (the options before –z) and linker options (the options following –z).
- The –z option must follow all source files and other compiler options on the command line or be specified with the C_OPTION environment variable.

All arguments that follow –z on the command line are passed on to the linker. These arguments can be linker command files, additional object files, linker options, or libraries. For example, to compile and link all the .c files in a directory, enter:

```
dspcl –sq *.c –z c.cmd –o prog.out –l rts25.lib
```

First, all of the files in the current directory that have a .c extension are compiled using the –s (interlist C and assembly code) and –q (run in quiet mode) options. Second, the linker links the resulting object files by using the c.cmd command file. The –o option names the output file, and the –l option names the run-time-support library.

The order in which the linker processes arguments is important. The compiler passes arguments to the linker in the following order:

1) Object filenames from the command line
2) Arguments following the –z option on the command line
3) Arguments following the –z option from the C_OPTION environment variable

4.3 Disabling the Linker (–c Shell Option)

You can override the –z option by using the –c shell option. The –c option is especially helpful if you specify the –z option in the C_OPTION environment variable and want to selectively disable linking with the –c option on the command line.

The –c linker option has a different function than, and is independent of, the –c shell option. By default, the compiler uses the –c linker option when you use the –z option. This tells the linker to use C linking conventions (autoinitialization of variables at runtime). If you want to initialize variables at load time, use the –cr linker option following the –z option.

4.4 Linker Options

All command-line input following the –z shell option is passed to the linker as parameters and options. The following are options that control the linker along with detailed descriptions of their effects.

–a Produces an absolute, executable module. This is the default; if neither –a nor –r is specified, the linker acts as if –a is specified.

–ar Produces a relocatable, executable object module

–b Disables merging of symbolic debugging information

–c Autoinitializes variables at runtime. See section 6.8.4 on page 6-34, for more information.

–cr Initializes variables at load time. See section 6.8.5 on page 6-35, for more information.

–e *global_symbol* Defines a *global_symbol* that specifies the primary entry point for the output module

–f *fill_value* Sets the default fill value for holes within output sections; *fill_value* is a 16-bit constant

–g *global_symbol* Defines *global_symbol* as global even if the global symbol has been made static with the –h linker option

–h Makes all global symbols static

–heap *size* Sets heap size (for dynamic memory allocation) to *size* words and defines a global symbol that specifies the heap size. Default is 1K words.

–i *directory* Alters the library-search algorithm to look in *directory* before looking in the default location. This option must appear before the –l linker option. The directory must follow operating system conventions.

–l *libraryname* (lower case L) Names an archive library file or linker command filename as linker input. The *libraryname* is an archive library name and must follow operating system conventions.

–m *filename* Produces a map or listing of the input and output sections, including holes, and places the listing in *filename.* The *filename* must follow operating system conventions.

–n Ignores all fill specifications in memory directives. Use this option in the development stage of a project to avoid generating large .out files, which can result from using memory directive fill specifications.

–o *filename*	Names the executable output module. The *filename* must follow operating system conventions. If the –o option is not used, the default filename is a.out.
–q	Requests a quiet run (suppresses the banner and progress information)
–r	Retains relocation entries in the output module
–s	Strips symbol table information and line number entries from the output module
–stack *size*	Sets the C system stack size to *size* words and defines a global symbol that specifies the stack size. Default is 1K words.
–u *symbol*	Places the unresolved external symbol *symbol* into the output module's symbol table
–v0	Generates version 0 COFF format
–v1	Generates version 1 COFF format
–v2	Generates version 2 COFF format
–w	Displays a message when an undefined output section is created
–x	Forces rereading of libraries. Resolves back references

For more information on linker options, see the *Linker Description* chapter in the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide.*

4.5 Controlling the Linking Process

Regardless of the method you choose for invoking the linker, special requirements apply when linking C programs. You must:

- Include the compiler's run-time-support library
- Specify the type of initialization
- Determine how you want to allocate your program into memory

This section discusses how these factors are controlled and provides an example of the standard default linker command file.

For more information about how to operate the linker, see the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide.*

4.5.1 Linking With Run-Time-Support Libraries

You must link all C programs with a run-time-support library. The library contains standard C functions as well as functions used by the compiler to manage the C environment. You must use the –l linker option to specify which TMS320C2x/C2xx/C5x run-time-support library to use. The –l option also tells the linker to look at the –i options and then the C_DIR environment variable to find an archive path or object file. To use the –l linker option, type on the command line:

dsplnk {**–c** | **–cr**} *filenames* **–l** *libraryname*

Generally, you should specify the library as the last name on the command line because the linker searches libraries for unresolved references in the order that files are specified on the command line. If any object files follow a library, references from those object files to that library are not resolved. You can use the –x linker option to force the linker to reread all libraries until references are resolved. Whenever you specify a library as linker input, the linker includes and links only those library members that resolve undefined references.

Three versions of the standard run-time-support library are included with the compiler: rts25.lib for TMS320C2x programs, rts2xx.lib for TMS320C2xx programs, and rts50.lib for TMS320C5x programs.

You must link all C programs with an object module called *boot.obj*. When a C program begins running, it must execute boot.obj first. The boot.obj file contains code and data to initialize the run-time environment; the linker automatically extracts boot.obj and links it when you use –c or –cr and include rts25.lib, rts2xx.lib or rts50.lib in the link.

Note: The _c_int0 Symbol

One important function contained in the run-time-support library is _c_int0. The symbol _c_int0 is the starting point in boot.obj; if you use the –c or –cr linker option, _c_int0 is automatically defined as the entry point for the program. If your program begins running from reset, you should set up the reset vector to branch to _c_int0 so that the processor executes boot.obj first.

The boot.obj module contains code and data for initializing the run-time environment. The module performs the following tasks:

1) Sets up the stack
2) Processes the run-time initialization table and autoinitializes global variables (when using the –c option)
3) Calls main
4) Calls exit when main returns

Chapter 7 describes additional run-time-support functions that are included in the library. These functions include ANSI C standard run-time support.

4.5.2 Specifying the Type of Initialization

The C compiler produces data tables for initializing global variables. Section 6.8.3, *Initialization Tables*, on page 6-33 discusses the format of these tables. These tables are in a named section called *.cinit*. The initialization tables are used in one of the following ways:

- Global variables are initialized at *run time*. Use the –c linker option (see section 6.8.4, *Autoinitialization of Variables at Run Time*, on page 6-34).
- Global variables are initialized at *load time*. Use the –cr linker option (see section 6.8.5, *Initialization of Variables at Load Time*, on page 6-35).

When you link a C program, you must use either the –c or –cr linker option. These options tell the linker to select initialization at run time or load time. When you compile and link programs, the –c linker option is the default. If used, the –c linker option must follow the –z option. (See section 4.2, *Invoking the Linker With the Compiler Shell*, on page 4-4. The following list outlines the linking conventions used with –c or –cr:

- The symbol _c_int0 is defined as the program entry point; it identifies the beginning of the C boot routine in boot.obj. When you use –c or –cr, _c_int0 is automatically referenced, ensuring that boot.obj is automatically linked in from the run-time-support library.
- The *.cinit* output section is padded with a termination record so that the loader (load time initialization) or the boot routine (run time initialization) knows when to stop reading the initialization tables.
- When using initializing at load time (the –cr linker option), the following occur:
  - The linker sets the symbol cinit to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag (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.
- When autoinitializing at run time (–c linker option), the linker defines the symbol cinit as the starting address of the .cinit section. The boot routine uses this symbol as the starting point for autoinitialization.

4.5.3 Specifying Where to Allocate Sections in Memory

The compiler produces relocatable blocks of code and data. These blocks, called *sections*, are allocated in memory in a variety of ways to conform to a variety of system configurations.

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 4–2 summarizes the sections.

Table 4–2. Sections Created by the Compiler

(a) Initialized sections

Name	Contents
.cinit	Tables for explicitly initialized global and static variables
.const	String literals, and global and static const variables that are explicitly initialized
.switch	Jump tables for large switch statements
.text	Executable code and floating-point constants

(b) Uninitialized sections

Name	Contents
.bss	Global and static variables
.stack	Software stack
.sysmem	Dynamic memory area for malloc functions (heap)

When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM. Though the .const section can be placed in ROM because it is never written to, it must be configured as data memory because of how it is accessed (see subsection 6.1.3, *Allocating .const to Program Memory*, on page 6-5, for details). See section 6.1.1, *Sections*, on page 6-3, for a complete description of how the compiler uses these sections. The linker provides MEMORY and SECTIONS directives for allocating sections.

The following table shows the type of memory and the page designation each section type requires:

Section	Type of Memory	Page
.text	ROM or RAM	0
.cinit	ROM or RAM	0
.const	ROM or RAM	1
.switch	ROM or RAM	0
.bss	RAM	1
.stack	RAM	1
.sysmem	RAM	1

For more information about allocating sections into memory, refer to the linker chapter of the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

4.5.4 A Sample Linker Command File

Example 4–1 shows a typical linker command file that links a C program. The command file in this example is named link.cmd and lists several linker options:

–c Tells the linker to use autoinitialization at run time.

–m Tells the linker to create a map file; the map file in this example is named example.map.

–o Tells the linker to create an executable object module; the module in this example is named example.out.

Example 4–1. An Example of a Linker Command File

```
/****************************************************/
/        Linker command file link.cmd
/****************************************************/
-c                  /* ROM autoinitialization model */
-m example.map      /* Create a map file            */
-o example.out      /* Output file name             */
main.obj            /* First C module               */
sub.obj             /* Second C module              */
asm.obj             /* Assembly language module     */
-l rts25.lib        /* Runtime-support library      */
-l matrix.lib       /* Object library               */
MEMORY
{
   PAGE 0 : PROG: origin = 30h,    length = 0EFD0h
   PAGE 1 : DATA: origin = 800h    length - 0E800h
}
SECTIONS
{
    .text   > PROG  PAGE 0
    .cinit  > PROG  PAGE 0
    .switch > PROG  PAGE 0
    .bss    > DATA  PAGE 1
    .const  > DATA  PAGE 1
    .sysmem > DATA  PAGE 1
    .stack  > DATA  PAGE 1
}
```

Next, the command file lists all the object files to be linked. This C program consists of two C modules, main.c and sub.c, which were compiled and assembled to create two object files called main.obj and sub.obj. This example also links in an assembly language module called asm.obj.

One of these files must define the symbol *main* because boot.obj calls main as the start of your C program. All of these single object files are linked.

Finally, the command file lists all the object libraries that the linker must search. (The libraries are specified with the –l linker option.) Because this is a C program, the run-time-support library (rts25.lib, rts2xx.lib, or rts50.lib) must be included. This program uses several routines from an archive library called matrix.lib, so it is also named as linker input. Note that only the library members that resolve undefined references are linked.

To link the program, enter:

```
dsplnk link.cmd
```

The MEMORY directive and possibly the SECTIONS directive might require modification to work with your system. Refer to the linker chapter of the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide* for information on these directives.

Chapter 5

TMS320C2x/C2xx/C5x C Language

The TMS320C2x/C2xx/C5x C compiler supports the C language standard that was developed by a committee of the American National Standards Institute (ANSI) to standardize the C programming language.

ANSI C supersedes the de facto C standard that is described in the first edition of *The C Programming Language* by Kernighan and Ritchie. The ANSI standard is described in the American National Standard for Information Systems—Programming Language C X3.159–1989. The second edition of *The C Programming Language* is based on the ANSI standard. ANSI C encompasses many of the language extensions provided by current C compilers and formalizes many previously unspecified characteristics of the language.

Topic **Page**

5.1 Characteristics of TMS320C2x/C2xx/C5x C Language 5-2
5.2 Data Types 5-4
5.3 Register Variables 5-6
5.4 Pragma Directives 5-7
5.5 The asm Statement 5-9
5.6 Creating Global Register Variables 5-10
5.7 Initializing Static and Global Variables 5-12
5.8 Compatibility with K&R C 5-14
5.9 Compiler Limits 5-16

5.1 Characteristics of TMS320C2x/C2xx/C5x C Language

The ANSI standard identifies certain features of the C language that are affected by characteristics of the target processor, run-time environment, or host environment. For efficiency or practicality, these characteristics can differ among standard compilers. This section describes how these characteristics are implemented for the TMS320C2x/C2xx/C5x C compiler.

The following list identifies all such cases and describes the behavior of the TMS320C2x/C2xx/C5x C compiler in each case. Each description also includes a reference to more information. Many of the references are to the formal ANSI standard or to the second edition of *The C Programming Language* by Kernighan and Ritchie (K&R).

5.1.1 Identifiers and Constants

- The first 100 characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct for identifiers. These characteristics apply to all identifiers, internal and external. (ANSI 3.1.2, K&R A2.3)
- The source (host) and execution (target) character sets are assumed to be ASCII. There are no multibyte characters. (ANSI 2.2.1, K&R A12.1)
- Hex or octal escape sequences in character or string constants may have values up to 32 bits. (ANSI 3.1.3.4, K&R A2.5.2)
- Character constants with multiple characters are encoded as the last character in the sequence. For example,

  `'abc' == 'c'` (ANSI 3.1.3.4, K&R A2.5.2)

5.1.2 Data Types

- For information about the representation of data types, see section 5.2, *Data Types*, on page 5-5. (ANSI 3.1.2.5, K&R A4.2)
- The type size_t, which is the result of the *sizeof* operator, is unsigned int. (ANSI 3.3.3.4, K&R A7.4.8)
- The type ptrdiff_t, which is the result of pointer subtraction, is int. (ANSI 3.3.6, K&R A7.7)

5.1.3 Conversions

- Float-to-integer conversions truncate toward zero. (ANSI 3.2.1.3, K&R A6.3)
- Pointers and integers can be freely converted. (ANSI 3.3.4, K&R A6.6)

5.1.4 Expressions

- When two signed integers are divided and either is negative, the quotient is negative, and the sign of the remainder is the same as the sign of the numerator. The slash mark (/) is used to find the quotient and the percent symbol (%) is used to find the remainder. For example:

```
10 / −3 == −3,    −10 / 3 == −3
10 % −3 == 1,     −10 % 3 == −1
```

  (ANSI 3.3.5, K&R A7.6)

- A right shift of a signed value is an arithmetic shift; that is, the sign is preserved. (ANSI 3.3.7, K&R A7.8)

5.1.5 Declarations

- The *register* storage class is effective for all character, short, integer, and pointer types. (ANSI 3.5.1, K&R A8.1)
- Structure members are not packed into words (with the exception of bit fields). Each member is aligned on a 16-bit word boundary. (ANSI 3.5.2.1, K&R A8.3)
- A bit field defined as an integer is signed. Bit fields are packed into words beginning at the low-order bits, and do not cross word boundaries. (ANSI 3.5.2.1, K&R A8.3)

5.1.6 Preprocessor

The preprocessor ignores any unsupported #pragma directive. (ANSI 3.8.6, K&R A12.8)

The following pragmas *are* supported:

- CODE_SECTION
- DATA_SECTION
- FUNC_EXT_CALLED

For more information on pragmas, see section 5.4 on page 5-7.

5.2 Data Types

- All integral types (char, short, int, and their unsigned counterparts) are equivalent types and are represented as 16-bit binary values.
- Long and unsigned long types are represented as 32-bit binary values.
- Signed types are represented in 2s-complement notation.
- The type char is a signed type, equivalent to int.
- Objects of type enum are represented as 16-bit values; the type enum is equivalent to int in expressions.
- All floating-point types (float, double, and long double) are equivalent and are represented in the TMS320C2x/C2xx/C5x's 32-bit floating-point format.
- Longs and floats are stored in memory with the least significant word at the lower address.

Table 5–1 lists the size, representation, and range of each scalar data type.

Table 5–1. TMS320C2x/C2xx/C5x C Data Types

Type	Size	Representation	Range Minimum	Range Maximum
char, signed char	16 bits	ASCII	–32 768	32 767
unsigned char	16 bits	ASCII	0	65 535
short	16 bits	2s complement	–32 768	32 767
unsigned short	16 bits	binary	0	65 535
int, signed int	16 bits	2s complement	–32 768	32 767
unsigned int	16 bits	binary	0	65 535
long, signed long	32 bits	2s complement	–2 147 483 648	2 147 483 647
unsigned long	32 bits	binary	0	4 294 967 295
enum	16 bits	2s complement	–32768	32 767
float	32 bits	TMS320C2x/C2xx/C5x	1.19 209 290e–38	3.4 028 235e+38
double	32 bits	TMS320C2x/C2xx/C5x	1.19 209 290e–38	3.4 028 235e+38
long double	32 bits	TMS320C2x/C2xx/C5x	1.19 209 290e–38	3.4 028 235e+38
pointers	16 bits	binary	0	0xFFFF

Many of the range values are available as standard macros in the header file limits.h, which is supplied with the compiler. For more information, see section 7.2.4, *Limits (float.h and limits.h)*, on page 7-6.

Note: TMS320C2x/C2xx/C5x Byte Is 16 Bits

By ANSI C definition, the sizeof operator yields the number of *bytes* required to store an object. ANSI further stipulates that when sizeof is applied to char, the result is 1. Since the TMS320C2x/C2xx/C5x char is 16 bits (to make it separately addressable), a byte is also 16 bits. This yields results you may not expect; for example, sizeof (int) = = 1 (*not* 2). TMS320C2x/C2xx/C5x bytes and words are equivalent (16 bits).

5.3 Register Variables

The C compiler uses up to two register variables within a function. You must declare the register variables in the argument list or in the first block of the function. Register declarations in nested blocks are treated as normal variables.

The compiler uses AR6 and AR7 for register variables:

- AR6 is assigned to the first register variable.
- AR7 is assigned to the second variable.

The *address* of the variable is placed into the allocated register to simplify access. Thus, 16-bit types (char, short, int, and pointers) may be used as register variables.

Setting up a register variable at run time requires approximately four instructions per register variable. To use this feature efficiently, use register variables only if the variable will be accessed more than twice.

The optimizer also creates register variables, but it uses them in a different way.

5.4 Pragma Directives

Pragma directives tell the compiler's preprocessor how to treat functions. The TMS320C2x/C2xx/C5x C compiler supports the following pragmas:

- CODE_SECTION
- DATA_SECTION
- FUNC_EXT_CALLED

Two of these pragmas use the arguments *func* and *symbol*. These arguments must have file scope; that is, you cannot define or declare them inside the body of a function. You must specify the pragma outside the body of a function, and it must occur before any declaration, definition, or reference to the *func* or *symbol* argument. If you do not follow these rules, the compiler issues a warning.

5.4.1 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in a section named *section name*. The syntax of the pragma is:

#pragma CODE_SECTION (*symbol*, *"section name"***);**

The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section.

Example 5–1 demonstrates the use of the CODE_SECTION pragma.

Example 5–1. Using the CODE_SECTION Pragma

(a) C source file

```
#pragma CODE_SECTION(fn, "my_sect")

int fn(int x)
{
   return c;
}
```

(b) Assembly source file

```
    .file   "CODEN.c"
    .sect   "my_sect"
    .global _fn
    .sym    _fn,_fn,36,2,0
    .func   3
```

5.4.2 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in a section named *section name*. The syntax of the pragma is:

#pragma DATA_SECTION (*symbol*, “*section name*”**);**

The DATA_SECTION pragma is useful if you have data objects that you want to link into an area separate from the .bss section.

Example 5–2 demonstrates the use of the DATA_SECTION pragma.

Example 5–2. Using the DATA_SECTION Pragma

(a) C source file

```
#pragma DATA_SECTION(bufferB, ”my_sect”)
char bufferA[512];
char bufferB[512];
```

(b) Assembly source file

```
            .global _bufferA
            .bss     _bufferA,512,4
            .global _bufferB
_bufferB:   .usect  ”my_sect”,512,4
```

5.4.3 The FUNC_EXT_CALLED Pragma

When you use the –pm option, the compiler uses program-level optimization. When you use this type of optimization, the compiler removes any function that is not called, directly or indirectly, by main. You might have C functions that are called by hand-coded assembly instead of main.

The FUNC_EXT_CALLED pragma specifies to the optimizer to keep these C functions or any other functions that these C functions call. These functions act as entry points into C.

The pragma must appear before any declaration or reference to the function that you want to keep. The syntax of the pragma is:

#pragma FUNC_EXT_CALLED (*func***);**

The argument *func* is the name of the C function that you do not want removed.

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. See section 3.3.2, *Optimization Considerations When Using Mixing C and Assembly*, on page 3-8.

5.5 The asm Statement

The TMS320C2x/C2xx/C5x C compiler can embed ’C6000 assembly language instructions or directives directly into the assembly language output of the compiler. This capability is an extension to the C language—the *asm* statement. The asm statement provides access to hardware features that C cannot provide. The asm statement is syntactically like a call to a function named asm, with one string constant argument:

asm(”*assembler text***”);**

The compiler copies the argument string directly into your output file. The assembler text must be enclosed in double quotes. All of the usual character string escape codes retain their definitions. For example, you can insert a .string directive that contains quotes:

```
asm(”STR: .string \”abc\””);
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line of code inside the quotes must begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, the assembler detects it. For more information about the assembly language statements, see the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User’s Guide*.

The asm statements do not follow the syntactic restrictions of normal C statements. Each can appear as a statement or a declaration, even outside of blocks. This is useful for inserting directives at the very beginning of a compiled module.

Note: Avoid Disrupting the C Environment With asm Statements

Be careful not to disrupt the C environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome.

Be especially careful when you use the optimizer with asm statements. Although the optimizer cannot remove asm statements, it can significantly rearrange the code order near them and cause undesired results.

5.6 Creating Global Register Variables

The TMS320C2x/C2xx/C5x compiler extends the C language by adding a special convention to the register keyword to allow the allocation of global registers. In this special case, the register keyword is treated as a storage class modifier. The declaration must appear before any function definitions. This special declaration has the form:

register *type* **AR6**
or
register *type* **AR7**

The two registers R6 and R7 are normally save-on-entry registers; *type* cannot be float or long. When you use the allocation declaration at file level, the register is permanently reserved from any other use by the optimizer and code generator for that file. You cannot assign an initial value to the register. You can use a #define statement to assign a meaningful variable name to the register and use the variable normally; for example:

```
register struct data_struct *AR6
#define data_pointer (AR6)

data_pointer->element;
data_pointer++;
```

5.6.1 When to Use a Global Register Variable

There are two reasons that you would be likely to use a global register variable:

- You are using a global variable throughout your program, and it would significantly reduce code size and execution speed to assign this variable to a register permanently.
- You are using an interrupt service routine that is executed so frequently that it would significantly reduce execution speed if the routine did not have to save and restore the register(s) it uses every time it is executed.

You should consider carefully the implications of assigning a global register variable. Registers are a precious resource to the compiler, and using this feature indiscriminately may result in less efficient code.

You should also carefully consider how code with a globally declared register variable interacts with other code, including library functions, that does not recognize the restriction placed on the register.

5.6.2 Avoiding Corrupting Register Values

Because the two registers you are allowed to use for global register variables are normally save-on-entry registers, a normal function call and return does not affect the value in the register and neither does a normal interrupt. However, when you mix code that has a globally declared register variable with code that does not have the register reserved, it is possible for the value in the register to become corrupted. To avoid corrupting the register values, you must follow these rules:

- You cannot access a global register variable in an interrupt service routine unless you recompile all code, including all libraries, to reserve the register.
- Functions that alter global register variables cannot be called by functions that are not aware of the global register. You must be careful if you pass a pointer to a function as an argument. If the passed function alters the global register variable and the called function saves the register, the value in the register is corrupted.
- You must save the global register on entry into a module that uses it, and you must restore the register at exit.
- The longjmp() function restores global register variables to the values they had at the setjmp() location. If this presents a problem in your code, you must unarchive the source for longjmp from rts.src and modify it.

5.6.3 Disabling the Compiler From Using AR6 and AR7

The −r*register* shell option and the corresponding −g*register* option for the optimizer and code generator (if you are invoking the tools individually) prevent the compiler from using the named *register*. This lets you reserve the named register in modules (such as the run-time-support libraries) that do not have the global register variable declaration if you need to compile the modules to prevent some of the above occurrences.

You can disable the compiler’s use of AR6 and AR7 completely so that you can use AR6 and/or AR7 in your interrupt functions without preserving them. If you disable the compiler from using AR6 and AR7, you must compile all code with the –r option(s) and rebuild the run-time-support library. For example, the following command rebuilds the rts25.lib library to not use AR6 and AR7:

```
dspmk −rAR6 −rAR7 −o −v25 rts.src −l rts25.lib
```

5.7 Initializing Static and Global Variables

The ANSI C standard specifies that static and global (extern) variables without explicit initializations must be initialized to 0 before the program begins running. This task is typically performed when the program is loaded. Because the loading process depends heavily on the specific environment of the target application system, the compiler itself makes no provision for preinitializing variables. It is up to your application to fulfill this requirement.

If your loader does not preinitialize variables, you can use the linker to preinitialize the variables to 0 in the object file. For example, in the linker command file, use a fill value of 0 in the .bss section:

```
SECTIONS
   {
      ...
      .bss: {} = 0x00;
      ...
   }
```

Because the linker writes a complete load image of the zeroed .bss section into the output COFF file, this method can have the unwanted effect of significantly increasing the size of the output file.

5.7.1 Initializing Static and Global Variables With the const Type Qualifier

Static and global variables with the type qualifier *const* are handled differently than other types of static and global variables.

The *const* static and global variables without explicit initializations are similar to other static and global variables because they may not be preinitialized to 0 (for the same reasons discussed above). For example:

```
const int zero;        /* may not be initialized to zero */
```

However, const, global, and static variables' initializations are different because they are declared and initialized in a section called .const. For example:

```
const int zero = 0;   /*    guaranteed to be zero       */
```

which corresponds to an entry in the .const section:

```
       .sect    .const
_zero
       .word    0
```

The feature is particularly useful for declaring a large table of constants because neither time nor space is wasted at system startup to initialize the table. Additionally, the linker can be used to place the .const section in ROM.

5.7.2 Accessing I/O Port Space

The ioport keyword enables access to the I/O port space of the TM320C2x/C2xx/C5x devices. The keyword has the form:

ioport *type* **port***hex_num*

ioport is the keyword that indicates this is a port variable.

type must be char, short, int, or the unsigned variable.

port *hex_num* refers to the port number. The *hex_num* is a hexadecimal number.

All declarations of port variables must be done at the file level. Port variables declared at the function level are not supported.

For example, the following code declares the I/O port as unsigned port 10h, writes a to port 10h, then reads port 10h into b:

```
ioport unsigned port10; /* variable to access I/O port 10h */

int func ()
{
    ...

    port10 = a;         /* write a to port 10h             */
    ...

    b = port10;         /* read port 10h into b            */
    ...
}
```

The use of port variables is not limited to assignments. Port variables can be used in expressions like any other variable. Following are examples:

```
call(port10);    /* read port 10h and pass to call          */

a = port10 + b;  /* read port 10h, add b, assign to a       */

port10 += a;     /* read port 10h, add a, write to port 10h */
```

5.8 Compatibility with K&R C

The ANSI C language is a superset of the de facto C standard defined in Kernighan and Ritchie's *The C Programming Language*. Most programs written for other non-ANSI compilers correctly compile and run without modification.

There are subtle changes, however, in the language that can affect existing code. Appendix C in *The C Programming Language* (second edition, referred to in this manual as K&R) summarizes the differences between ANSI C and the first edition's C standard (the first edition is referred to in this manual as K&R C).

To simplify the process of compiling existing C programs with the 'C6000 ANSI C compiler, the compiler has a K&R option (–pk) that modifies some semantic rules of the language for compatibility with older code. In general, the –pk option relaxes requirements that are stricter for ANSI C than for K&R C. The –pk option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, –pk simply liberalizes the ANSI rules without revoking any of the features.

The specific differences between the ANSI version of C and the K&R version of C are as follows:

- The integral promotion rules have changed regarding promoting an unsigned type to a wider signed type. Under K&R C, the result type was an unsigned version of the wider type; under ANSI, the result type is a signed version of the wider type. This affects operations that perform differently when applied to signed or unsigned operands; namely, comparisons, division (and mod), and right shift:

```
unsigned short u;
int i;
if (u < i) ...    /* SIGNED comparison, unless –pk used
*/
```

- ANSI prohibits combining two pointers to different types in an operation. In most K&R compilers, this situation produces only a warning. Such cases are still diagnosed when –pk is used, but with less severity:

```
int *p;
char *q = p; /* error without –pk, warning with –pk */
```

  Even without –pk, a violation of this rule is a code-E (recoverable) error. You can use –pe, which converts code-E errors to warnings, as an alternative to –pk.

- External declarations with no type or storage class (only an identifier) are illegal in ANSI but legal in K&R:

```
a;              /* illegal unless –pk used */
```

- ANSI interprets file scope definitions that have no initializers as *tentative definitions*. In a single module, multiple definitions of this form are fused together into a single definition. Under K&R, each definition is treated as a separate definition, resulting in multiple definitions of the same object (and usually an error). For example:

```
int a;
int a;               /* illegal if –pk used, OK if not */
```

  Under ANSI, the result of these two declarations is a single definition for the object a. For most K&R compilers, this sequence is illegal, because a is defined twice.

- ANSI prohibits, but K&R allows, objects with external linkage to be redeclared as static:

```
extern int a;
static int a;        /* illegal unless –pk used */
```

- Unrecognized escape sequences in string and character constants are explicitly illegal under ANSI but ignored under K&R:

```
char c = '\q';       /* same as 'q' if –pk, error if not */
```

- ANSI specifies that bit fields must be of type integer or unsigned. With –pk, bit fields can be legally defined with any integral type. For example:

```
struct s
{
   short f : 2;      /* illegal unless –pk used */
};
```

- K&R syntax allows a trailing comma in enumerator lists:

```
enum { a, b, c, };   /* illegal unless –pk used */
```

- K&R syntax allows trailing tokens on preprocessor directives:

```
#endif NAME          /* illegal unless –pk used */
```

5.9 Compiler Limits

Due to the variety of host systems that the TMS320C2x/C2xx/C5x C compiler supports and the limitations of some of these systems, the compiler might not be able to successfully compile source files that are excessively large or complex. Most of these conditions are detected by the parser. When the parser detects such a condition, it issues a code-I diagnostic message indicating the condition that caused the failure. Usually, the message also specifies the maximum value for whatever limit was exceeded. The code generator also has compilation limits, but fewer than the parser.

In general, exceeding any compiler limit prevents continued compilation, so the compiler aborts immediately after printing the error message. Simplify the program to avoid exceeding a compiler limit.

Many compiler tables have no absolute limits and are limited only by the amount of memory available in the host system. Table 5–2 specifies the limits that are absolute. All of the absolute limits equal or exceed those required by the ANSI C standard.

On smaller host systems such as PCs, the optimizer may run out of memory. If this occurs, the optimizer terminates and the shell continues compiling the file with the code generator. This results in a file compiled with no optimization. The optimizer compiles one function at a time, so the most likely cause of this is a large or extremely complex function in your source module. To correct the problem, your options are:

- ❑ Do not optimize the module in question.
- ❑ Identify the function that caused the problem and break it down into smaller functions.
- ❑ Extract the function from the module and place it in a separate module that can be compiled without optimization so that the remaining functions can be optimized.

Table 5–2. Absolute Compiler Limits

Description	Limits	
Filename length	512 characters	
Source line length	16K characters	(See Note 1)
Length of strings built from # or ##	512 characters	(See Note 2)
Macros predefined with– d	64	
Macro parameters	32 parms	
Macro nesting	32 levels	(See Note 3)
#include search paths	64 paths	(See Note 4)
#include file nesting	64 levels	
Conditional inclusion (#if) nesting	64 levels	
Nesting of struct, union, or prototype declarations	20 levels	
Function parameters	48 parms	
Array, function, or pointer derivations on a type	12 derivations	
Aggregate initialization nesting	32 levels	
Static initializers	1500 per initialization (approximately)	
Local initializers	150 levels (approximately)	
Nesting of if statements, switch statements, and loops	1500 per initialization (approximately)	
Global symbols	2000 - - - PCs 10000 - - All others (See Note 5)	
Block scope symbols visible at any point	500 . . . PCs 1000 . . All others	

Notes:
1) This limit reflects the number of characters after splicing of \ lines. This limit also applies to any single macro definition or invocation.
2) This limit reflects the number of characters before concatenation. All other character strings are unrestricted.
3) This limit includes argument substitutions.
4) This limit includes –i and C_DIR directories.
5) May be further limited by available system memory.

Table 5–2. Absolute Compiler Limits (Continued)

Description	Limits
Number of unique string constants	400 - - - - PCs 1000 - - - All others
Number of unique floating-point constants	400 . . PCs 1000 . . All others

Notes:
1) This limit reflects the number of characters after splicing of \ lines. This limit also applies to any single macro definition or invocation.
2) This limit reflects the number of characters before concatenation. All other character strings are unrestricted.
3) This limit includes argument substitutions.
4) This limit includes –i and C_DIR directories.
5) May be further limited by available system memory.

Chapter 6

Run-Time Environment

This chapter describes the TMS320C2x/C2xx/C5x C run-time environment. To ensure successful execution of C programs, it is critical that all run-time code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C code.

Topic **Page**

6.1	**Memory Model**	**6-2**
6.2	**Register Conventions**	**6-9**
6.3	**Function Structure and Calling Conventions**	**6-14**
6.4	**Interfacing C With Assembly Language**	**6-19**
6.5	**Interrupt Handling**	**6-25**
6.6	**Integer Expression Analysis**	**6-28**
6.7	**Floating-Point Expression Analysis**	**6-30**
6.8	**System Initialization**	**6-31**

6.1 Memory Model

The TMS320C2x/C2xx/C5x C compiler treats memory as two linear blocks of program memory and data memory:

- **Program memory** contains executable code.
- **Data memory** contains external variables, static variables, and the system stack.

Each block of code or data generated by a C program is placed into a contiguous block in the appropriate memory space.

Note: The Linker Defines the Memory Map

The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces.

For example, you can use the linker to allocate global variables into fast internal RAM or to allocate executable code into internal ROM. You can allocate each block of code or data individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although you can access physical memory locations with C pointer types).

6.1.1 Sections

The compiler produces relocatable blocks of code and data called *sections*. The sections can be allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about COFF sections, see the introductory COFF information in the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

The TMS320C2x/C2xx/C5x compiler creates the following types of sections:

- **Initialized sections** contain data tables or executable code. The C compiler creates the following initialized sections:
  - The **.text section** contains all the executable code as well as floating-point constants.
  - The **.cinit section** contains tables for initializing variables and constants.
  - The **.const section** contains string constants, and the declaration and initialization of global and static variables (qualified by *const*) that are explicitly initialized.
  - The **.switch section** contains tables for switch statements.
- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at run time for creating and storing variables. The compiler creates the following uninitialized sections:
  - The **.bss section** reserves space for global and static variables. When you specify the –c linker option, at program startup time, the C boot routine copies data out of the .cinit section (which can be in ROM) and stores it in the .bss section.
  - The **.stack section** allocates memory for the C system stack. This memory passes arguments to functions and to allocates space for local variables.
  - The **.sysmem section** reserves space for dynamic memory allocation. The reserved space is utilized by calloc, malloc, and realloc functions. If a C program does not use these functions, the compiler does not create the .sysmem section.

The assembler creates the default sections .text, .bss, and .data. The C compiler, however, does not use the .data section. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see sections 5.4.1, *The CODE_SECTION Pragma*, on page 5-7 and 5.4.2, *The DATA_SECTION Pragma*, on page 5-8).

The linker takes the individual sections from different modules and combines sections with the same name to create output sections. The complete program is made up of these output sections, which include the assembler's .data section. You can place these output sections anywhere in the address space as needed in order to meet system requirements.

The .text, .cinit, and .switch sections are usually linked into either ROM or RAM, and must be in program memory (page 0). The .const section can also be linked into either ROM or RAM but must be in data memory (page 1). The .bss, .stack, and .sysmem sections should be linked into RAM, and must be in data memory (page 1). The following table shows the type of memory and page designation each section type requires:

Section	Type of Memory	Page
.text	ROM or RAM	0
.cinit	ROM or RAM	0
.switch	ROM or RAM	0
.const	ROM or RAM	1
.bss	RAM	1
.stack	RAM	1
.sysmem	RAM	1

For more information about allocating sections into memory, see the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

6.1.2 C System Stack

The C compiler uses the software stack to:

- Allocate local variables
- Pass arguments to functions
- Save the processor status
- Save function return address
- Save temporary results
- Save registers

The run-time stack grows up from low addresses to higher addresses. The compiler uses two auxiliary registers to manage this stack:

AR1 is the **stack pointer** (SP). It points to the current top of the stack *or* to the word that follows the current top of the stack.

AR0 is the **frame pointer** (FP). It points to the beginning of the current frame. Each function invocation creates a new frame at the top of the stack, from which local and temporary variables are allocated.

The C environment manipulates these registers automatically. If you write assembly language routines that use the run-time stack, be sure to use these registers correctly. For more information about using these registers, see section 6.2, *Register Conventions*, on page 6-9. For more information about the stack, see section 6.3, *Function Calling Conventions*, on page 6-14.

The linker sets the stack size, creates a global symbol, __STACK_SIZE, and assigns it a value equal to the stack size in bytes. The default stack size is 1K bytes. You can change the size of the stack at link time by using the –stack option with the linker. For more information, see section 4.4, *Linker Options*, on page 4-6.

At system initialization, SP is set to a designated address for the bottom of the stack. This address is the first location in the .stack section. Since the position of the stack depends on where the .stack section is allocated, the actual address of the stack is determined at link time. If you allocate the stack as the last section in memory (highest address), the stack has unlimited space for growth (within the limits of system memory).

Note: Stack Overflow

The compiler provides no means to check for stack overflow during compilation or at run time. A stack overflow will disrupt the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow.

6.1.3 Allocating .const to Program Memory

If your system configuration does not support allocating an initialized section such as .const to data memory, then you must allocate the .const section to load in program memory and run in data memory. At boot time, copy the .const section from program to data memory. The following sequence shows how you can perform this task:

First, modify the boot routine using the following steps:

1) Extract boot.asm from the source library:

```
dspar –x rts.src boot.asm
```

2) Edit boot.asm and change the CONST_COPY flag to 1:

```
CONST_COPY .set  1
```

3) Assemble boot.asm:

```
dspa –v<target> boot.asm
```

4) Archive the boot routine into the object library:

```
dspar –r rts<target>.lib boot.obj
```

Next, link with a linker command file that contains the following entries:

```
MEMORY
{
   PAGE 0 : PROG : ...
   PAGE 1 : DATA : ...
}

SECTIONS
{
   ...
   .const  : load = PROG PAGE 1, run = DATA PAGE 1
             {
                /* GET RUN ADDRESS   */
                 __const_run = .;
                /* MARK LOAD ADDRESS */
                 *(.c_mark)
                /* ALLOCATE .const   */
                 *(.const)
                /* COMPUTE LENGTH    */
                 __const_length = .- __const_run;
              }
   ...
}
```

Your linker command file can substitute the name PROG with the name of a memory area on page 0, and DATA with the name of a memory area on page 1. The rest of the command file must use the names as above. The code in boot.asm that is enabled when you change CONST_COPY to 1 depends on the linker command file using these names in this manner. To change any of the names, you must edit boot.asm and change the names in the same way.

6.1.4 Dynamic Memory Allocation

Dynamic memory allocation is not a standard part of the C language. The run-time-support library supplied with the TMS320C2x/C2xx/C5x compiler contains several functions (such as malloc, calloc, and realloc) that allow you to allocate memory dynamically for variables at run time.

Memory is allocated from a global pool, or heap, that is defined in the .sysmem section. You can set the size of the .sysmem section by using the –heap *size* option with the linker command. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 0x400 bytes. For more information on the –heap option, see section 4.4, *Linker Options,* on page 4-6.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers) and the memory pool is in a separate section (.sysmem); therefore, the dynamic memory pool can have a size limited only by the amount of available memory in your system. To conserve space in the .bss section, you can allocate large arrays from the heap instead of defining them as global or static. For example, instead of a definition such as:

```
struct big table[100];
```

use a pointer and call the malloc function:

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

6.1.5 Initialization of Variables

The C compiler produces code that is suitable for use as firmware in a ROM-based system. In such a system, the initialization tables in the .cinit section (used for initialization of globals and statics) are stored in ROM. At system initialization time, the C boot routine copies data from these tables (in ROM) to the initialized variables in .bss (RAM).

In situations where a program is loaded directly from an object file into memory and run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file instead of from ROM and perform the initialization directly at load time (instead of at run time). You can specify this to the linker by using the –cr linker option. For more information on system initialization, see section 6.8, *System Initialization*, on page 6-31.

6.1.6 Allocating Memory for Static and Global Variables

A unique, contiguous space is allocated for each external or static variable that is declared in a C program. The linker determines the address of the space. The compiler ensures that space for these variables is allocated in multiples of words so that each variable is aligned on a word boundary.

The C compiler expects global variables to be allocated into data memory by reserving space for them in .bss. Variables declared in the same module are allocated into a single, contiguous block of memory.

6.1.7 Field/Structure Alignment

When the compiler allocates space for a structure, it allocates as many words as are needed to hold all of the structure's members. In an array of structures, each structure begins on a word boundary.

All nonfield types are aligned on word boundaries. Fields are allocated as many bits as requested. Adjacent fields are packed into adjacent bits of a word, but they do not overlap words. If a field would overlap into the next word, the entire field is placed into the next word. Fields are packed as they are encountered with the least significant bits of the structure word are filled first.

6.1.8 Character String Constants

In C, a character string constant is used in one of the following ways:

- To initialize an array of characters. For example:

  ```
  char s[ ] = "abc";
  ```

  When a string is used as an initializer, it is simply treated as an initialized array, each character being a separate initializer. For more information about initialization, see section 6.8, *System Initialization*, on page 6-31.

- In an expression. For example:

  ```
  strcpy (s, "abc");
  ```

  When a string is used in an expression, the string itself is defined in the .const section with the .string assembler directive, along with a unique label that points to the string the terminating 0 byte is also included. For example, the following lines define the string abc, and the terminating byte (the label SL5 points to the string):

  ```
        .sect     .const
  SL5   .string   "abc", 0
  ```

  String labels have the form **SL***n*, where *n* is a number assigned by the compiler to make the label unique. The number begins at 0 and is increased by 1 for each string defined. All strings used in a source module are defined at the end of the compiled assembly language module.

  The label SL*n* represents the address of the string constant. The compiler uses this label to reference the string in the expression.

  If the same string is used more than once within a source module, the string is not duplicated in memory. All uses of an identical string constant share a single definition of the string.

  Because strings are stored in the .const section (possibly in ROM) and shared, it is bad practice for a program to modify a string constant. The following code is an example of incorrect string use:

  ```
  const char *a = "abc"
  a[1] = 'x';         /* Incorrect! */
  ```

6.2 Register Conventions

Strict conventions associate specific registers with specific operations in the C environment. If you plan to interface an assembly language routine to a C program. You must understand and follow these register conventions.

The register conventions dictate both how the compiler uses registers and how values are preserved across function calls. There are two types of register variable registers, save on call and save on entry. The distinction between these two types of register variable registers is the method by which they are preserved across calls. It is the called function's responsibility to preserve save-on-entry register variables, and the calling function's responsibility to preserve save-on-call register variables.

The compiler uses registers differently, depending on whether or not you use the optimizer (–o option). The optimizer uses additional registers for register variables (variables defined to reside in a register rather than in memory). However, the conventions for preserving registers across function calls are identical with or without the optimizer.

The following table summarizes how the compiler uses the TMS320C2x/C2xx/C5x registers and shows which registers are defined to be preserved across function calls.

Table 6–1. Register Use and Preservation Conventions

(a) TMS320C2x, TMS320C2xx, and TMS320C5x conventions

Register	Usage	Preserved by Call
AR0	Frame pointer	Yes
AR1	Stack pointer	Yes
AR2	Local variable pointer	No
AR2–AR5	Expression analysis	No
AR6–AR7	Register variables	Yes
Accumulator	Expression analysis/return values	No
P	Expression analysis	No
T	Expression analysis	No

(b) TMS320C5x-only conventions

Register	Usage	Preserved by Call
INDX	Shadows AR0	Yes
ACCB	Expression analysis	No
TREG1	Expression analysis	No
BRCR	Loop counter	No
PASR/PAER	Block repeat registers	No

6.2.1 Status Register Fields

Table 6–2 shows all of the status fields used by the compiler. Presumed value is the value the compiler expects in that field upon entry to, or return from, a function. A dash in this column indicates that the compiler does not expect a particular value. The modified column indicates if code generated by the compiler ever modifies this field.

Table 6–2. Status Register Fields

(a) TMS320C2x, TMS320C2xx and TMS320C5x fields

Field	Name	Presumed Value	Modified
ARP	Auxiliary-register pointer	1	Yes
C	Carry	–	Yes
DP	Data page	–	Yes
OV	Overflow	–	Yes
OVM	Overflow mode	0	No
PM	Product shift mode	0	No
SXM	Sign-extension mode	–	Yes
TC	Test-control bit	–	Yes

(b) TMS320C5x-only fields

Field	Name	Presumed Value	Modified
BRAF	Block-repeat active flag	–	Yes
NDX	Index-register enable bit	0	No
TRM	Enable multiple TREGs	0	No

6.2.2 Stack Pointer, Frame Pointer, and Local Variable Pointer

The compiler creates and uses its own software stack for saving the function return addresses, allocating local (automatic) variables, and passing arguments to functions. The internal hardware stack is not used to save the function return address except in cases where the compiler can be certain the call depth (the number of function invocations on the stack at the same time) does not exceed eight levels. When a function requires local storage, it creates its own working space (local frame) from the stack. The local frame is allocated during the function’s entry sequence and deallocated during the return sequence.

Three registers, the stack pointer (SP), the frame pointer (FP), and the local variable pointer (LVP), manage the stack and the local frame.

Register AR1 is dedicated as the stack pointer. The compiler uses the SP in the conventional way: the stack grows towards higher addresses, and the SP points to the next available word on the stack.

Register AR0 is dedicated as the frame pointer. The FP points to the beginning of the local frame for the current function. The first word of the local frame, which is directly pointed to by the FP, is used as a temporary memory location to allow register-to-register transfers and is essential to creating reentrant C functions.

Register AR2 is dedicated as the local variable pointer. All objects stored on the local frame, including arguments, are referenced indirectly through the LVP. See section 6.3.4, *Accessing Arguments and Local Variables*, on page 6-18, for information on how the LVP is used to access objects on the frame.

6.2.3 The TMS320C5x INDX Register

On the TMS320C5x, the *0+ addressing mode adds the INDX register, not AR0, into the AR register indicated by the ARP (auxiliary-register pointer-field of status register ST0). The compiler presumes the NDX bit of the status register PMST is 0, which means that changes to the register AR0 are shadowed in INDX. This also means that the INDX register must be preserved across calls just as AR0 is. For code executing with NDX = 0, preserving AR0 preserves INDX as well. If, however, you write an assembly routine that changes the NDX bit to 1, both AR0 and INDX must be preserved explicitly.

6.2.4 Register Variables

Register variables are local variables or compiler temporaries defined to reside in a register rather than in memory. The way the compiler uses registers for register variables differs depending on whether you use the optimizer.

6.2.4.1 Register Variables When the Optimizer is Not Used

When the optimizer is not used, the compiler allocates registers for up to two variables declared with the register keyword. You must declare the variables in the argument list or in the first block of the function. Register declarations in nested blocks are treated as normal variables.

The compiler uses AR6 and AR7 for these register variables. AR6 is allocated to the first variable, and AR7 is allocated to the second register variable.

The *address* of the variable is placed into the allocated register to simplify access. Only 16-bit types (char, short, int, and pointers) can be used as register variables.

Setting up a register variable at run time requires approximately four instructions per register variable. To use this feature efficiently, use register variables only if the variable is accessed more than twice.

6.2.4.2 Register Variables When the Optimizer is Used

When the optimizer is used, all user register declarations are ignored. The optimizer decides which variables or compiler temporaries are allocated to registers. The optimizer allocates the variables, not their addresses, directly to registers. The optimizer can allocate the registers AR5, AR6, and AR7 for register variables. Because AR5 is not preserved across function calls, it is used for variables that overlap any calls.

Because the register use for variables depends on whether or not you use the optimizer, you should avoid writing code that depends on specific registers allocated to specific variables.

Note: Using AR6 and AR7 as Global Register Variables

If you have disabled the compiler from using AR6 and AR7 with the –r option, AR6 and AR7 are not available for use as register variables. See section 5.6.3, *Disabling the Compiler From Using AR6 and AR7* on page 5-11, for more information.

6.2.5 Expression Registers

The compiler uses registers not being used for register variables to evaluate expressions and store temporary results. The contents of the expression registers are not preserved across function calls. Any register that is being used for temporary storage when a call occurs is saved to the local frame before the function is called. This prevents the called function from saving and restoring expression registers.

6.2.6 Return Values

When a value of any scalar type (integer, pointer, or floating point) is returned from a function, the value is placed in the accumulator when the function returns.

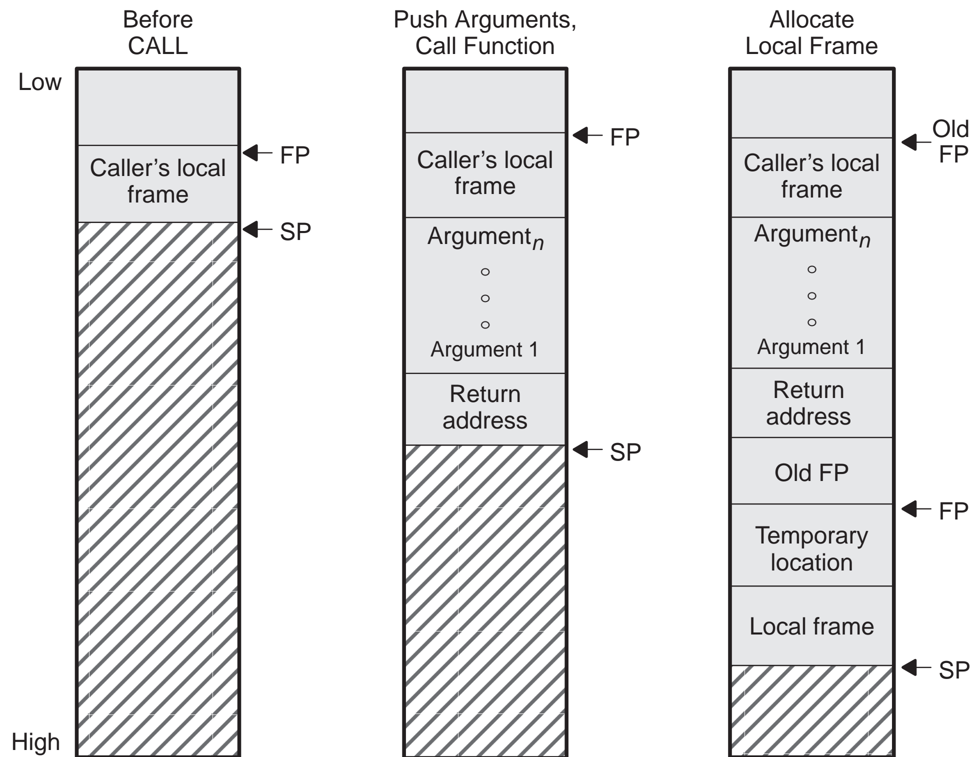
Sixteen-bit types (char, short, int, or pointer) are loaded into the accumulator with correct sign extension.

6.3 Function Structure and Calling Conventions

The C compiler imposes a strict set of rules on function calls. Except for special run-time-support functions, any function that calls or is called by a C function must follow these rules. Failure to adhere to these rules can disrupt the C environment and cause a program to fail.

Figure 6–1 illustrates a typical function call. In this example, parameters are passed to the function, and the function uses local variables. This example also shows allocation of a local frame for the called function. Functions that have *no* arguments passed on the stack *and no* local variables do not allocate a local frame.

Figure 6–1. Stack Use During a Function Call



6.3.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function). Be aware that the ARP must be set to AR1.

1) The caller pushes the arguments on the stack in reverse order (the rightmost declared argument is pushed first, and the leftmost is pushed last). This places the leftmost argument at the top of the stack when the function is called.
2) The caller (parent) calls the function (child).
3) The caller presumes that upon return from the function, the ARP will be set to AR1.
4) When the called function is complete, the caller pops the arguments off the stack with the following instruction:

   `SBRK n` (*n* is the number of argument words that were pushed)

6.3.2 How a Called Function Responds

A called function (child function) must perform the following tasks. On function entry, the ARP is presumed to be set to SP (AR1).

1) Pop the return address off the hardware stack and push it onto the software stack.
2) Push the FP onto the software stack.
3) Allocate the local frame.
4) If the function modifies AR6 or AR7, push them on the stack. Any other registers may be modified without preserving them.
5) Execute the code for the function.
6) If the function returns a scalar value, place in the accumulator. Load 16-bit integer and pointer return values into the accumulator with the proper sign extension.
7) Set the ARP to AR1.
8) Restore AR6 and/or AR7, if they were saved.
9) Deallocate the local frame.
10) Restore the FP.
11) Copy the return address from the software stack and push it onto the hardware stack.
12) Return.

Example 6–1 is an example of TMS320C2x code that performs the tasks listed in section 4.3.2.

Example 6–1. TMS320C2x Code as a Called Function

```
                       ; presume ARP = AR1 (SP)
    POPD   *+          ; pop return address, push on software stack
    SAR    AR0,*+      ; push AR0 (FP)
    SAR    AR1,*       ; *SP = SP
    LARK   AR0,SIZE    ; FP  = size of frame
    LAR    AR0,*0+     ; FP  = SP, SP += size ==> allocate frame
    SAR    AR6,*+      ; push AR6
    SAR    AR7,*+      ; push AR7

    ...                ; code for the function

    MAR    *,AR1       ; set ARP = SP
    MAR    *-          ; point to saved AR7
    LAR    AR7,*-      ; pop AR7
    LAR    AR6,*-      ; pop AR6
    SBRK   SIZE+1      ; deallocate frame, point to saved FP
    LAR    AR0,*-      ; pop FP
    PSHD   *           ; push return address on hardware stack
    RET                ; return
```

6.3.3 Special Cases for a Called Function

There are four special cases for a called function:

- ❑ Returning a structure
- ❑ Not moving the return address to the software stack
- ❑ Not allocating a frame
- ❑ Using the TMS320C5x RETD instruction

These cases are explained in the following sections.

6.3.3.1 Returning a structure

If the function returns a structure, the caller (parent function) allocates space for the structure and then passes the address of the return space to the called function (child function) as an additional and final argument on the stack. To return a structure, the called function then copies the structure to the memory block pointed to by this argument. If the caller does not use the return value, the value of the argument is 0. This directs the called function not to copy the return structure.

In this way, the caller can accurately tell the called function where to return the structure. For example, in the statement s = f( ), where s is a structure and f is a function that returns a structure, the caller can simply pass the address of s as the last argument and call f. Function f then copies the return structure directly into s, performing the assignment automatically.

You must be careful to properly declare functions that return structures, both at the point where they are called (so the caller passes the address of the return space as the last argument) and where they are defined (so the function knows to copy the result).

6.3.3.2 Not moving the return address to the software stack

If this function calls no other functions, or if the only functions called are from a list of run-time-support functions the compiler knows will not exceed a call depth of 8, then there is no need to pop the return address off of the hardware stack and push it on the software stack. Steps 2 and 12 of section 6.3.2, *How a Called Function Responds*, on page 6-15, are omitted.

6.3.3.3 Not allocating a local frame

If there are no local variables, no arguments, no use of the temporary location pointed to by AR0, the code is not being compiled to run under the debugger, and the function does not return a structure, there is no need to allocate a local frame. Steps 3, 4, 10, and 11 of section 6.3.2, *How a Called Function Responds,* on page 6-15, are omitted. If the return address is saved on the software stack and no registers are saved on the stack, step 10 is replaced by a MAR *– to point the SP to the saved return address.

6.3.3.4 Using the TMS320C5x RETD *instruction*

The debugger expects the compiler to generate the frame as described above. When generating code for the 'C5x that will not be run under the debugger, the compiler switches steps 2 and 3 of section 6.3.2, *How a Called Function Responds*, on page 6-15, as well as steps 11 and 12. This occurs because a PUSHD instruction cannot go in the delay slots (the two words following a delayed instruction) of a RETD, but a LAR AR0,* can. For the case above, the last three instructions would be changed as follows:

```
PSHD   *-       ; push return address on hardware stack
RETD            ; return delayed
LAR    AR0,*    ; restore FP
NOP             ; fill delay slots of RETD
```

If the return address is not saved on the stack, which means a PSHD will not be generated, the RETD will be moved two words from the end of the function.

6.3.4 Accessing Arguments and Local Variables

In general terms, the compiler performs the first local access, initializing LVP (AR2) by loading it with the offset of the variable relative to the FP, then a MAR *0+ instruction adds in the FP. Subsequent accesses are performed by adding or subtracting the difference between the address of the current local variable LVP is pointing to and the next one. Because the LVP is not preserved across calls, it must be reinitialized after a call.

Arguments are always at negative offsets from the FP, and locals are always at positive offsets from the FP.

6.4 Interfacing C with Assembly Language

The following are ways to use assembly language with C code:

- Use separate modules of assembled code and link them with compiled C modules (see section 6.4.1, *Using Assembly Language Modules With C Code*). This is the most versatile method.
- Use inline assembly language, embedded directly in the C source (see section 6.4.2, *Using Inline Assembly Language*, on page 6-22).
- Use assembly language variables in C source (see section 6.4.3, *Accessing Assembly Language Variables From C*, on page 6-23.
- Modify the assembly language code that the compiler produces (see section 6.4.4, *Modifying Compiler Output*, on page 6-24).

6.4.1 Using Assembly Language Modules With C Code

Interfacing C with assembly language functions is straightforward if you follow the calling conventions defined in section 6.3, *Function Structure and Calling Conventions*, on page 6-14 and the register conventions defined in section 6.2, *Register Conventions,* on page 6-9. C code can access variables and call functions defined in assembly language, and assembly code can access C variables and call C functions.

Follow these guidelines to interface assembly language and C:

- All functions, whether they are written in C or assembly language, must follow the register conventions outlined in section 6.2, *Register Conventions*, on page 6-9.
- You must preserve any dedicated registers modified by a function. Dedicated registers include:

  AR0 (FP)
  AR1 (SP)
  AR6
  AR7
  INDX (TMS320C5x only)

  If you use the stack normally, you do not need to explicitly preserve the SP. In other words, you are free to use the stack inside a function as long as you pop everything you pushed before your function exits.

  You can use all other registers freely used without preserving their contents.

When using only the ’C5x: if your assembly routine does not change the NDX bit of status register PMST from 0 to 1, then the INDX register shadows AR0; preserving AR0 preserves INDX as well. If your routine does change the NDX bit, then both AR0 and INDX must be preserved explicitly.

- If you change any of the status register fields for which Table 6–2 on page 6-11 shows a presumed value, you must ensure that value is restored. Be especially careful that you ensure that the ARP is AR1.
- Interrupt routines must save *all* the registers they use. (For more information, see section 6.5, *Interrupt Handling*, on page 6-25.)
- When you call a C function from assembly language, push any arguments onto the stack in reverse order. Pop them off after calling the function.
- When you call C functions, remember that only the dedicated registers listed above are preserved. C functions can change the contents of any other register.
- Longs and floats are stored in memory with the least significant word at the lower address.
- Functions must return values in the accumulator. 16-bit integer values and pointers must be loaded in the accumulator with proper sign extension.
- No assembly module should use the .cinit section for any purpose other than autoinitialization of global variables. The C startup routine in boot.c assumes that the .cinit section consists *entirely* of initialization tables. Disrupting the tables by putting other information in .cinit causes unpredictable results.
- The compiler adds an underscore (_) to the beginning of all identifiers (that is, labels). In assembly language modules, you must use an underscore prefix for all objects that are to be accessible from C. For example, a C object named x is called _x in assembly language. Identifiers that are used only in assembly language modules can use any name that does not begin with an underscore without conflicting with a C identifier.
- Any object or function declared in assembly language that is accessed or called from C must be declared with the .global directive in the assembler. This declares the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C function or object from assembly language, declare the C object with .global. This creates an undeclared external reference that the linker resolves.

Example 6–2 illustrates a C function called main, which calls an assembly language function called asmfunc. The asmfunc function takes its single argument, adds it to the C global variable called gvar, and returns the result.

Example 6–2. An Assembly Language Function

(a) C program

```
extern int asmfunc(); /* declare external asm function  */
int gvar;             /* define global variable         */

main()
{
   int i;

   i = asmfunc(i);    /* call function normally         */
}
```

(b) Assembly language program

```
_asmfunc:
   POPD      *+        ; Move return address to C stack
   SAR  AR0, *+        ; Save FP
   SAR  AR1, *         ; Save SP
   LARK AR0, 1         ; Size of frame
   LAR  AR0, *0+, AR2  ; Set up FP and SP

   LDPK _gvar          ; Point to gvar
   SSXM                ; Set sign extension
   LAC  _gvar          ; Load gvar
   LARK AR2, -3        ; Offset of argument
   MAR  *0+            ; Point to argument
   ADD  *, AR0         ; Add arg to gvar
   SACL _gvar          ; Save in gvar

   LARP AR1            ; Pop off frame
   SBRK 2
   LAR  AR0, *         ; Restore frame pointer
   PSHD *              ; Move return addr to C2x stack
   RET
```

In the C program in Example 6–2, the extern declaration of asmfunc is optional, since the return type is int. Like C functions, you need to declare assembly functions only if they return noninteger values or pass noninteger parameters.

Further, in Example 6–2 it is not necessary to move the return address from the hardware stack to the software stack, because asmfunc makes no calls. The code is in the example to illustrate how it is accomplished.

6.4.2 Using Inline Assembly Language

Within a C program, you can use the *asm statement* to inject a single line of assembly language into the assembly language file that the compiler creates. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code.

Note: Using the asm Statement

The asm statement lets you access features of the hardware that would be otherwise inaccessible from C. When you use the asm statement, be careful not to disrupt the C environment. The compiler does not check or analyze the inserted instructions.

Inserting jumps or labels into C code may produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.

Do not change the value of a C variable. You can, however, safely read the current value of any variable.

Do not use the asm statement to insert assembler directives that would change the assembly environment.

The asm statement is also useful for inserting comments in the compiler output. You can simply start the assembly code string with an asterisk (*) as shown below:

```
asm(”**** this is an assembly language comment”);
```

6.4.3 Accessing Assembly Language Variables From C Code

It is sometimes useful for a C program to access variables defined in assembly language. Accessing uninitialized variables from the .bss section or from a named section is straightforward:

1) Use the .bss or .usect directive to define the variable.
2) Use the .global directive to make the definition external.
3) Precede the name with an underscore in assembly language.
4) In C, declare the variable as extern, and access it normally.

Example 6–3 shows how you can access a variable defined in .bss.

Example 6–3. Accessing a Variable Defined in .bss From C

(a) C program

```
extern int var;      /* External variable          */
var = 1;             /* Use the variable           */
```

(b) Assembly language program

```
* Note the use of underscores in the following lines

    .bss      _var,1    ; Define the variable
    .global   _var      ; Declare it as external
```

You may not always want a variable to be in the .bss section. A common situation is a lookup table defined in assembly language that you do not want to put in RAM. In this case, you must define a pointer to the object and access it indirectly from C.

You must first define the object. It is helpful, but not necessary, to put it in its own initialized section. Declare a global label that points to the beginning of the object, and then the object can be linked anywhere into the memory space. To access it in C, you must declare the object as *extern* and not precede it with an underscore. Then you can access the object normally.

Example 6–4 shows an example that accesses a variable that is not defined in .bss.

Example 6–4. Accessing a Variable Not Defined in .bss From C

(a) C program

```
extern float sine[];    /* This is the object         */
f = sine[4];            /* Access sine as normal array*/
```

(b) Assembly language program

```
    .global   _sine     ; Declare variable as external
    .sect  ”sine_tab”   ; Make a separate section
_sine:       ; The table starts here
    .float 0.0
    .float 0.015987
    .float 0.022145
```

6.4.4 Modifying Compiler Output

You can inspect and change the assembly language output produced by the compiler by compiling the source and then editing the output file before assembling it. The C interlist utility is useful for inspecting compiler output. For information on the interlist utility, see section 2.7, *Using the Interlist Utility,* on page 2-33. The warnings in section 6.4.2 about disrupting the C environment also apply to the modification of compiler output.

6.5 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C code without disrupting the C environment. When the C environment is initialized, the startup routine does not enable or disable interrupts. Interrupts are disabled if the system is initialized via a hardware reset. If your system uses interrupts, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C environment and are easily incorporated with asm statements or by callling an assembly language function.

6.5.1 General Points About Interrupts

- An interrupt routine can perform any task performed by any other function, including accessing global variables, allocating local variables, and calling other functions.
- When an interrupt routine is entered, the run-time-support function I$$SAVE is called to save the complete context of the interrupted function. All registers are saved. Upon return from the interrupt routine, the run-time-support function I$$REST is called to restore the environment and return to the interrupted function.
- The name c_int0 is the C entry point; this name is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the function main. Because it has no caller, c_int0 does not save any registers.
- To associate an interrupt routine with an interrupt, a branch must be placed in the appropriate interrupt vector. You can use the assembler and linker to do this by creating a simple table of branch instructions with the .sect assembler directive. For information on where the interrupt vector table is located, consult the user’s guide for the device you are targeting.

6.5.2 Using C Interrupt Routines

You can handle interrupts *directly* with C functions by using one of two conventions:

- Any function with the name c_int*d*, where *d* is a digit 0–9, is presumed to be an interrupt routine. The name c_int0 is reserved for the system reset interrupt; do not use this name for any other function. For example:

```
void c_int1()
{
  ...
}
```

- Or, you can use the interrupt keyword. For example:

```
interrupt void isr()
{
  ...
}
```

Using one of these conventions defines an interrupt routine. When the compiler encounters one of these routines, it generates code that allows the function to be activated from an interrupt trap. This method provides more functionality than the standard C signal mechanism. This does not prevent implementation of the signal function, but it does allow these functions to be written entirely in C.

When handling interrupts with C functions, remember the following:

- An interrupt routine must be of type void, and it should be declared with no arguments.
- The compiler does *not* save all the device registers. The compiler saves only those registers listed in Table 6–1.
- You must handle any special masking of interrupts via the IMR register. You can use inline assembly language to enable or disable the interrupts and modify the IMR register without corrupting the C environment.
- An interrupt routine can be called by normal C code, but it is inefficient to do so because all the registers are preserved by a calling C function.
- An interrupt routine can handle a single interrupt or multiple interrupts. The compiler does not generate code that is specific to a certain interrupt, except for c_int0, which is the system reset interrupt.
- None of the interrupt routines nor any of the functions they call can be compiled with the –oe shell option (optimizer option –j). The –oe option assumes that none of the functions in the module are interrupts, can be called by interrupts, or can be otherwise executed in an asynchronous manner, so compiling programs containing interrupt routines with this option negates their use.

6.5.3 Using Assembly Language Interrupt Routines

You can handle Interrupts with assembly language code as long as you follow the same register conventions the compiler. Keep the following points in mind:

- The word pointed to by the SP (AR1) may be in use by the compiler. It must be saved.
- The interrupt routine must preserve the registers from Table 6–1 and status bits from Table 6–2 on page 6-11 that it modifies.
- If the interrupt routine calls a C function, it must preserve *all* registers listed in Table 6–1 on page 6-10 that are not preserved by a call. Any other register can be modified by the C routine.
- Remember to precede the symbol name with an underscore. For example, refer to c_int0 as _c_int0.

6.5.4 TMS320C5x Shadow Register Capability

The TMS320C5x device automatically saves certain registers upon an interrupt trap in a set of internal shadow registers. See the *TMS320C5x User's Guide* for more information. If an interrupt is not nested (that is, does not reenable interrupts so that this interrupt routine is itself interruptible), then using the shadow register capability is the best way to preserve those registers.

If none of the interrupts you have written in C are nested, then you can take advantage of the shadow register capability by modifying an assembly time flag in the source of the I$$SAVE/I$$RESTORE routines that the compiler uses to preserve registers, as follows:

1) Unarchive the source from source library

```
dspar -x rts.src saverest.asm
```

2) Change the NEST flag in the source to 0

```
NEST  .set  0
```

3) Reassemble

```
dspa -v50 saverest.asm
```

4) Archive the new object file into the object library

```
dspar -r rts50.lib saverest.obj
```

6.6 Integer Expression Analysis

This section describes some special considerations for you to remember when evaluating integer expressions.

6.6.1 Arithmetic Overflow and Underflow

The TMS320C2x/C2xx/C5x produces a 32-bit result even when 16-bit values are used as data operands; thus, *arithmetic overflow and underflow cannot be handled in a predictable manner*. If code depends on a particular type of overflow/underflow handling, there is no assurance that this code will execute correctly. Generally, it is a good practice to avoid such code because it is not portable.

6.6.2 Integer Division and Modulus

The TMS320C2x/C2xx/C5x does not directly support integer division, so all division and modulus operations are performed through calls to run-time-support routines. These functions push the right-hand portion (divisor) of the operation onto the stack and then place the left-hand portion (dividend) into the 16 LSBs of the accumulator. The function places the result in the accumulator.

6.6.3 Long (32-Bit) Expression Analysis

Long expression analysis operations in C are performed with function calls that *do not* follow the standard C calling conventions. These functions work together with the compiler to maximize speed and minimize code space. The operations are:

- Left shift by a variable
- Right shift by a variable
- Division
- Modulus
- Multiplication

6.6.4 C Code Access to the Upper 16 Bits of 16-Bit Multiply

The following method provides access to the upper 16 bits of a 16-bit multiply in C language. For example:

- Signed results:

```
intm1, m2;
intresult;

result = ((long) m1 * (long) m2) >> 16;
```

- Unsigned results:

```
unsigned m1, m2;
unsigned result;

result = ((unsigned long) m1 * (unsigned long) m2) >>
16;
```

Both result statements are implemented by the compiler without making a function call to the 32-bit multiply routine.

6.7 Floating-Point Expression Analysis

The TMS320C2x/C2xx/C5x C compiler represents floating-point values as IEEE single-precision numbers. Both single-precision and double-precision floating-point numbers are represented as 32-bit values; there is no difference between the two formats.

The TMS320C2x/C2xx/C5x run-time-support library, rts.src, contains a custom-coded set of floating-point math functions that support:

- Addition, subtraction, multiplication, and division
- Comparisons (>, <, >=, <=, ==, !=)
- Conversions from integer or long to floating-point and floating-point to integer or long, both signed and unsigned
- Standard error handling

These functions *do not* follow standard C calling conventions. Instead, the compiler pushes the arguments onto the run-time stack and generates a call to a floating-point function. The function pops the arguments, performs the operation, and pushes the result onto the stack.

Some floating-point functions expect integer or long arguments or return integer or long values. For floating-point functions, all integers are passed and returned in the 16 LSBs of the accumulator, and all longs are passed and returned in all 32 bits of the accumulator.

6.8 System Initialization

Before you can run a C program, you must create the C run-time environment. The C boot routine performs this task using a function called c_int0. The run-time-support source library, rts.src, contains the source to this routine in a module named boot.asm.

To begin running the system, the c_int0 function can be branched to or called, but it is usually vectored to by reset hardware. You must link the c_int0 function with the other object modules. This occurs automatically when you use the –c or –cr linker option and include rts25.lib, rts2xx.lib, or rts50.lib as one of the linker input files.

When C programs are linked, the linker sets the entry point value in the executable output module to the symbol _c_int0. This does not, however, set the hardware to automatically vector to c_int00 at reset (see the).

The c_int0 function performs the following tasks to initialize the environment:

1) It defines a section called .stack for the system stack and sets up the initial stack pointers.

2) It initializes global variables by copying the data from the initialization tables in the .cinit section to the storage allocated for the variables in the .bss section. If you are initializing variables at load time (–cr option), a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see section 6.8.2, *Automatic Initialization of Variables*, on page 6-32.

3) It calls the function main to run the C program.

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C environment.

6.8.1 Run-time Stack

The run-time stack is allocated in a single contiguous block of memory and grows up from low addresses to higher addresses. Register AR1 usually points to the next available word in the stack (top of the stack plus one word). The compiler can use this word as a temporary memory location, so it must be saved by interrupt routines.

The code does not check to see if the run-time stack overflows. Stack overflow occurs when the stack grows beyond the limits of the memory space that was allocated for it. Be sure to allocate adequate memory for the stack.

The stack size can be changed at link time by using the –stack linker option on the linker command line and specifying the stack size as a constant directly after the option.

6.8.2 Automatic Initialization of Variables

Some global variables must have initial values assigned to them before a C program starts running. The process of retrieving these variables' data and intializing the variables with the data is called autoinitialization.

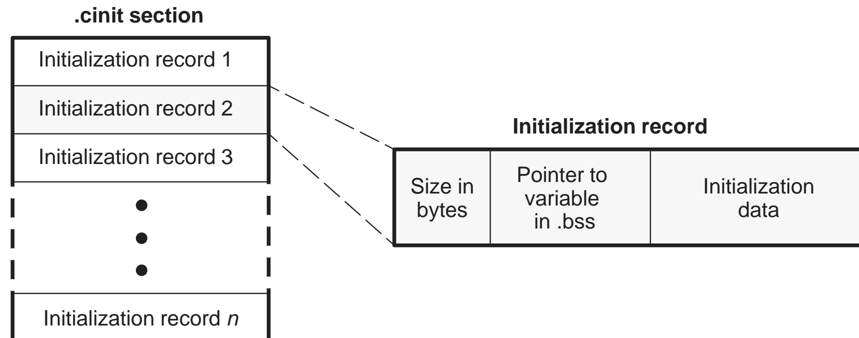
The compiler builds tables in a special section called .cinit that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine or a loader uses this table to initialize all the system variables.

Global variables are either autoinitialized at run time or at load time (see sections 6.8.4, *Autoinitialization of Variables at Run Time*, on page 6-34 and 6.8.5, *Initialization of Variables at Load Time*, on page 6-35).

6.8.3 Initialization Tables

The tables in the .cinit section consist of variable-size initialization records. Each variable that must be autoinitialized has a record in the .cinti section. Figure 6–2 shows the format of the .cinit section and the initialization records.

Figure 6–2. Format of Initialization Records in the .cinit Section



An initialization record contains the following information:

1) The first field (word 0) of an initialization record is the size (in words) of the initialization data for the variable.

2) The second field (word 1) contains the starting address of the area within the .bss section where the initialization data must be copied.

3) The third field (words 2 through *n*) contains the data that is copied into the .bss section to initialize the variable.

Each variable that must be autoinitialized has an initialization record. For example, suppose two initialized variables are defined in C as follows:

```
int    i = 23;
int    a[5] = { 1, 2, 3, 4, 5 };
```

The initialization tables appear as follows:

```
       .sect     ".cinit"     ; Initialization section
* Initialization record for variable i
       .word     1            ; Length of data (1 word)
       .word     _i           ; Address in .bss
       .word     23           ; Data to initialize i
* Initialization record for variable a
       .word     5            ; Length of data (5 words)
       .word     _a           ; Address in .bss
       .word     1,2,3,4,5    ; Data to initialize a
```

The .cinit section must contain only initialization tables in this format. If you interface assembly language modules to your C program, do not use the .cinit section for any other purpose.

When you use the −c or −cr linker option, the linker combines the .cinit sections from all the C modules and appends a null word to the end of the composite .cinit section. This terminating record appears as a record with a size field of 0, marking the end of the initialization tables.

The const-qualified variables are initialized differently; see section 5.7.1, *Initializing Static and Global Variables With the const Type Qualifier*, on page 5-12.

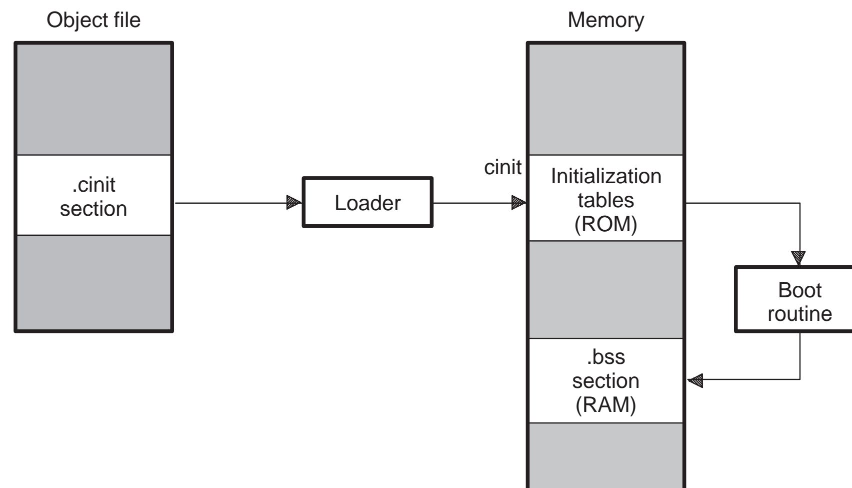
6.8.4 Autoinitialization of Variables at Run Time

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

Using this method, the .cinit section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 6–3 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Figure 6–3. Autoinitialization at Run Time



6.8.5 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the –cr option.

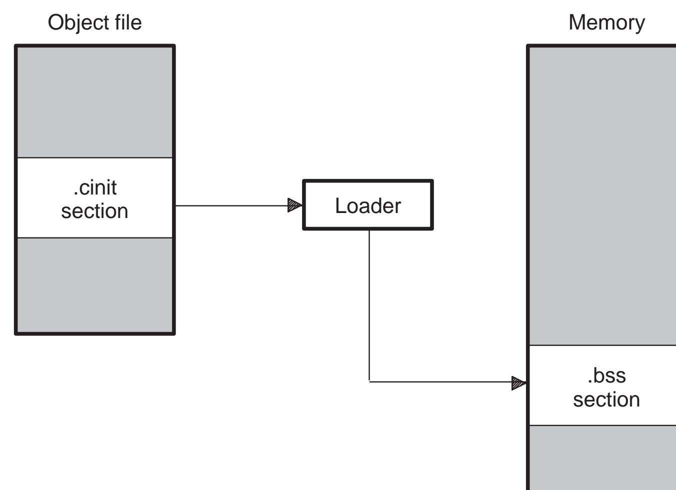
When you use the –cr linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to –1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run time initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the .cinit section in the object file
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory
- Understand the format of the initialization tables

Figure 6–4 illustrates the initialization of variables at load time.

Figure 6–4. Initialization at Load Time



Chapter 7

Run-Time-Support Functions

Some of the tasks that a C program performs (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C language itself. However, the ANSI C standard defines a set of run-time-support functions that perform these tasks. The TMS320C2x/C2xx/C5x C compiler implements the complete ANSI standard library except for those facilities that handle exception conditions and locale issues (properties that depend on local language, nationality, or culture). Using the ANSI standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI-specified functions, the TMS320C2x/C2xx/C5x run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests.

A library build utility is provided with the code generation tools that lets you create customized run-time-support libraries. The use of this utility is covered in Chapter 8, *Library-Build Utility*.

Topic **Page**

7.1 Libraries 7-2

7.2 Header Files 7-4

7.3 Summary of Run-Time-Support Functions and Macros 7-13

7.4 Descriptions of Run-Time-Support Functions and Macros 7-20

7.1 Libraries

The following libraries are included with the TMS320C2x/C2xx/C5x C compiler:

- ❑ rts25.lib, rts2xx.lib and rts50.lib—run-time-support object libraries. The libraries do not contain functions involving signals and locale issues. They do contain the following:
  - ANSI C standard library
  - System startup routine, _c_int00
  - Functions and macros that allow C to access specific instructions
- ❑ rts.src—run-time-support source library. The run-time-support object libraries are built from the C and assembly source contained in the rts.src library.

7.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved.

You should specify libraries *last* on the linker command line because the linker searches a library for unresolved references when it encounters the library on the command line. You can also use the –x linker option to force repeated searches of each library until the linker can resolve no more references.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

There is one header file, values.h, in rts.src. It is not a standard header, but allows you to customize the functions. It contains definitions necessary for recompiling the trigonometric and transcendental math functions.

7.1.2 Modifying a Library Function

You can inspect or modify library functions by using the archiver to extract the appropriate source file or files from the source libraries. For example, the following command extracts two source files:

```
dspar –x rts.src atoi.c strcpy.c
```

To modify a function, extract the source as in the previous example. Make the required changes to the code, recompile, and then reinstall the new object file or files into the library:

```
dspar –r rts25.lib atoi.obj strcpy.obj
```

You can also build a new library this way, rather than rebuilding back into rts25.lib. For more information about the archiver, see the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

7.1.3 Building a Library With Different Options

You can create a new library from rts.src by using the library-build utility, dspmk. For example, use this command to build a short, optimized run-time-support library:

```
dspmk ––u –o2 –x –ms rts.src –l rtsf.lib
```

The ––u option tells the dspmk utility to use the header files in the current directory, rather than extracting them from the source archive. The use of the optimizer (–o2) and inline function expansion (–x) options does not affect compatibility with code compiled with these options. The –ms option tells the compiler to ignore code speed and generate the shortest possible code.

For more information on the library-build utility, see Chapter 8.

7.2 Header Files

Each run-time-support function is declared in a *header file*. Each header file declares the following:

- A set of related functions (or macros)
- Any types that you need to use the functions
- Any macros that you need to use the functions

These are the header files that declare the run-time-support functions:

```
assert.h      limits.h      stddef.h
ctype.h       math.h        stdlib.h
errno.h       setjmp.h      string.h
float.h       stdarg.h      time.h
ioports.h
```

In order to use a run-time-support function, you must first use the #include preprocessor directive to include the header file that declares the function. For example, the isdigit function is declared by the ctype.h header. Before you can use the isdigit function, you must first include ctype.h:

```
#include  <ctype.h>
    .
    .
    .
    val = isdigit(num);
```

You can include headers in any order. You must include a header before you reference any of the functions or objects that it declares.

Sections 7.2.1, *Diagnostic Messages (assert.h)*, on page 7-5 through 7.2.12, *Time Functions (time.h)*, on page 7-11 describe the header files that are included with the C compiler. Section 7.3, *Summary of Run-Time-Support Functions and Macros*, on page 7-13, lists the functions that these headers declare.

7.2.1 Diagnostic Messages (assert.h)

The assert.h header defines the assert macro, which inserts diagnostic failure messages into programs at run time. The assert macro tests a run-time expression.

- If the expression is true (nonzero), the program continues running.
- If the expression is false, the macro outputs a message that contains the expression, the source file name, and the line number of the statement that contains the expression; then, the program terminates (using the abort function).

The assert.h header refers to another macro named NDEBUG (assert.h does not define NDEBUG). If you have defined NDEBUG as a macro name when you include assert.h, then assert is turned off and disabled. If NDEBUG *is not* defined, then assert is enabled.

The assert function is listed in Table 7–3 (a) on page 7-14.

7.2.2 Character-Typing and Conversion (ctype.h)

The ctype.h header declares functions that test (type) and convert characters.

The character-typing functions test a character to determine whether it is a letter, a printing character, a hexadecimal digit, etc. These functions return a value of *true* (a nonzero value) or *false* (0). Character-typing functions have names in the form is*xxx* (for example, *isdigit*).

The character conversion functions convert characters to lower case, upper case, or ASCII and return the converted character. Character-conversion functions have names in the form to*xxx* (for example, *toupper*).

The ctype.h header also contains macro definitions that perform these same operations. The macros run faster than the corresponding functions. The typing macros expand to a lookup operation in an array of flags (this array is defined in ctype.c). The macros have the same name as the corresponding functions, but each macro is prefixed with an underscore (for example, *_isdigit*). Use the function version, if an argument passed that has side effects.

The character typing and conversion functions are listed in Table 7–3 (b) page 7-14.

7.2.3 Error Reporting (errno.h)

Errors can occur in a math function if invalid parameter values are passed to the function or if the function returns a result that is outside the defined range for the type of the result. When this happens, a variable named *errno* is set to the value of one of the following macros:

- EDOM, for domain errors (invalid parameter)
- ERANGE, for range errors (invalid result)

C code that calls a math function can read the value of errno to check for error conditions. The errno variable is declared in errno.h and defined in errno.c.

7.2.4 Limits (float.h and limits.h)

The float.h and limits.h headers define macros that expand to useful limits and parameters of the TMS320C2x/C2xx/C5x's numeric representations. Table 7–1 and Table 7–2 list these macros and their limits.

Table 7–1. Macros That Supply Integer Type Range Limits (limits.h)

Macro	Value	Description
CHAR_BIT	16	Maximum number of bits for the smallest object that is not a bit field
SCHAR_MIN	–32 768	Minimum value for a signed char
SCHAR_MAX	32 767	Maximum value for a signed char
UCHAR_MAX	65 535	Maximum value for an unsigned char
CHAR_MIN	SCHAR_MIN	Minimum value for a char
CHAR_MAX	SCHAR_MAX	Maximum value for a char
SHRT_MIN	–32 768	Minimum value for a short int
SHRT_MAX	32 767	Maximum value for a short int
USHRT_MAX	65 535	Maximum value for an unsigned short int
INT_MIN	–32 768	Minimum value for an int
INT_MAX	32 767	Maximum value for an int
UINT_MAX	65 535	Maximum value for an unsigned int
LONG_MIN	–2 147 483 648	Minimum value for a long int
LONG_MAX	2 147 483 647	Maximum value for a long int
ULONG_MAX	4 294 967 295	Maximum value for an unsigned long int

Table 7–2. Macros That Supply Floating-Point Range Limits (float.h)

Macro	Value	Description
FLT_RADIX	2	Base or radix of exponent representation
FLT_ROUNDS	1	Rounding mode for floating-point addition (rounds toward integer)
FLT_DIG DBL_DIG LDBL_DIG	6	Number of decimal digits of precision for a float, double, or long double
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24	Number of base-FLT_RADIX digits in the mantissa of a float, double, or long double
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	–125	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128	Maximum integer such that FLT_RADIX raised to that power is a representative finite float, double, or long double
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290E-07F	Minimum positive float, double, or long double number *x* such that $1.0 + x \neq 1.0$
FLT_MIN DBL_MIN LDBL_MIN	1.17549435E-38F	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.40282347E+38F	Maximum positive float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–37	Minimum negative integer such that 10 raised to that power is in the range of normalized floats, doubles, or long doubles
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38	Maximum positive integer such that 10 raised to that power is in the range of finite floats, doubles, or long doubles

Key to prefixes:
FLT_ applies to type float.
DBL_ applies to type double.
LDBL_ applies to type long double.

7.2.5 Inport/Outport Macros (ioports.h)

The ioports.h header defines two macros and their associated functions, which are used to access the TMS320C2x/C2xx/C5x I/O ports. The macros are the easiest way to access the I/O ports and include:

- *inport*, a macro that reads a value from the specified port and returns the value via the pointer *ret*
- *outport*, a macro that writes a value to the specified port and has no return value

The functions include:

- *_inport x* , functions that access the port of the same number and return that value as an int. These functions follow this syntax:

  _inport *x***( )** where $0 \le x \le 15$

- *_outport<x>*, functions that access the port of the same number and have no return value. These functions follow this syntax:

  _outport *x***(***int value***)** where $0 \le x \le 15$

- *_in_port*, a function that reads an int from the specified port and returns the value the function follows this syntax:

  _in_port (int port)

- *_out_port* , a function that writes the output to the specified port and has no return value. This function follows this syntax:

  _out_port (int port, int value)

The _PSWITCH setting determines which method is used to implement the inport and outport macros. If _PSWITCH is set to 0 (the default), a switch statement is used that selects the correct _inport *x* or _outport *x* function. The 0 setting works well when the port number is a constant, because the compiler optimizes the switch statement into a simple call to the required function. However, if the port number is a variable, the entire switch remains as you wrote it. Therefore, when the port number is a variable, set _PSWITCH to 1 to call the correct _in_port or _out_port function. The macros always work, regardless of the value of _PSWITCH.

The inport and outport macros are listed in Table 7–3 (c) on page 7-14.

7.2.6 Floating-Point Math (math.h)

The math.h header defines several trigonometric, exponential, and hyperbolic math functions. These functions are listed in Table 7–3 (d) on page 7-15. The math functions expect double-precision floating-point arguments and return double-precision floating-point values. Except where all trigonometic functions use angles expressed as radians.

The math.h header also defines one macro named HUGE_VAL. The math functions use this macro to represent out-of-range values. When a function produces a floating-point return value that is too large to be represented, it returns HUGE_VAL instead.

For all math.h functions, domain and range errors are handled by setting errno to EDOM or ERANGE, as appropriate. The function input/outputs are rounded to the nearest legal value.

7.2.7 Nonlocal Jumps (setjmp.h)

The setjmp.h header defines a type and a macro and declares a function for bypassing the normal function call and return discipline. These include:

- *jmpbuf*, an array type suitable for holding the information needed to restore a calling environment
- *setjmp*, a macro that saves its calling environment in its jmp_buf argument for later use by the longjmp function
- *longjmp*, a function that uses its jmp_buf argument to restore the program environment

The nonlocal jump macro and function are listed in Table 7–3 (e) on page 7-16.

7.2.8 Variable Arguments (stdarg.h)

Some functions can have a variable number of arguments whose types can differ. Such functions are called *variable-argument functions*. The stdarg.h header declares macros and a type that help you to use variable-argument functions:

- The macros are *va_start*, *va_arg*, and *va_end*. These macros are used when the number and type of arguments may vary each time a function is called.
- The type *va_list* is a pointer type that can hold information for va_start, va_end, and va_arg.

A variable-argument function can use the macros declared by stdarg.h to step through its argument list at run time when the function knows the number and types of arguments actually passed to it. You must ensure that a call to a variable-argument function has visibility to a prototype for the function in order for the arguments to be handled correctly. The variable argument functions are listed in Table 7–3 (f) page 7-16.

7.2.9 Standard Definitions (stddef.h)

The stddef.h header defines types and macros. The types are:

- *ptrdiff_t*, a signed integer type that is the data type resulting from the subtraction of two pointers
- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator.

The macros are:

- *NULL*, a macro that expands to a null pointer constant(0)
- *offsetof(type, identifier)*, a macro that expands to an integer that has type size_t. The result is the value of an offset in bytes to a structure member (*identifier*) from the beginning of its structure (*type*).

These types and macros are used by several of the run-time-support functions.

7.2.10 General Utilities (stdlib.h)

The stdlib.h header defines a macro and types and declares several functions. The macro is named *RAND_MAX*, and it returns the largest value returned by the rand() function.The types are:

- *div_t*, a structure type that is the type of the value returned by the div function
- *ldiv_t*, a structure type that is the type of the value returned by the ldiv function

The functions are:

- Memory management functions that allow you to allocate and deallocate packets of memory. These functions can use 1K words of memory by default. You can change this amount at link time by invoking the linker with the –heap option.
- String conversion functions that convert strings to numeric representations

- Searching and sorting functions that allow you to search and sort arrays
- Sequence-generation functions that allow you to generate a pseudo-random sequence and allow you to choose a starting point for a sequence
- Program-exit functions that allow your program to terminate normally or abnormally
- Integer-arithmetic that is not provided as a standard part of the C language

The general utility functions are listed in Table 7–3 (g) on page 7-16.

7.2.11 String Functions (string.h)

The string.h header declares standard functions that perform the following tasks with character arrays (strings):

- Move or copy entire strings or portions of strings
- Concatenate strings
- Compare strings
- Search strings for characters or other strings
- Find the length of a string

In C, all character strings are terminated with a 0 (null) character. The string functions named str*xxx* all operate according to this convention. Additional functions that are also declared in string.h allow you to perform corresponding operations on arbitrary sequences of bytes (data objects), where a 0 value does not terminate the object. These functions are named mem*xxx*.

When you use functions that move or copy strings, be sure that the destination is large enough to contain the result. The string functions are listed in Table 7–3 (h) on page 7-17.

7.2.12 Time Functions (time.h)

The time.h header defines a macro and several types and declares functions that manipulate dates and time. The functions deal with several types of time:

- Calendar time represents the current date (according to the Gregorian calendar) and time.
- Local time is the calendar time expressed for a specific time zone.
- Daylight saving time is a variation of local time.

The time.h header declares one macro, *CLK_TCK*, which is the number per second of the value returned by the clock function.

The types are:

- *clock_t*, an arithmetic type that represents time
- *time_t*, an arithmetic type that represents time
- *tm, a* structure that holds the components of calendar time, called *broken-down time*. The structure has the following members:

```
int tm_sec;   /* seconds after the minute (0–59)   */
int tm_min;   /* minutes after the hour (0–59)     */
int tm_hour;  /* hours after midnight (0–23)       */
int tm_mday;  /* day of the month (1–31)           */
int tm_mon;   /* months since January (0–11)       */
int tm_year;  /* years since 1900 (0–99)           */
int tm_wday;  /* days since Saturday (0–6)         */
int tm_yday;  /* days since January 1 (0–365)      */
int tm_isdst; /* Daylight Saving Time flag         */
```

tm_isdst can have one of three values:

- A *positive* value if daylight saving time is in effect
- *Zero* if daylight saving time is not in effect
- A *negative* value if the information is not available

The time functions and macros are listed in Table 7–3 (i) on page 7-19.

Note: Customizing Time Functions

All of the time functions depend on the clock and time functions, which you must customize for your system.

7.3 Summary of Run-Time-Support Functions and Macros

Table 7–3 summarizes the run-time-support header files (in alphabetical order) provided with the TMS320C2x/C2xx/C5x ANSI C compiler. Most of the functions described are per the ANSI standard and behave exactly as described in the standard.

The functions and macros listed in Table 7–3 are described in detail in section 7.4, *Description of Run-Time-Support Functions and Macros* on page 7-20 . For a complete description of a function or macro, see the indicated page.

A superscripted number is used in the following descriptions to show exponents. For example, x^y is the equivalent of x to the power y.

Table 7–3. Summary of Run-Time-Support Functions and Macros

(a) Error message macro (assert.h)

Macro	Description	Page
void **assert**(int expr);‡	Inserts diagnostic messages into programs	7-22

(b) Character typing and conversion functions (ctype.h)

Function	Description	Page
int **isalnum**(int c);	Tests c to see if it is an alphanumeric-ASCII character	7-34
int **isalpha**(int c);	Tests c to see if it is an alphabetic-ASCII character	7-34
int **isascii**(int c);	Tests c to see if it is an ASCII character	7-34
int **iscntrl**(int c);	Tests c to see if it is a control character	7-34
int **isdigit**(int c);	Tests c to see if it is a numeric character	7-34
int **isgraph**(int c);	Tests c to see if it is any printing character except a space	7-34
int **islower**(int c);	Tests c to see if it is a lowercase alphabetic ASCII charcter	7-34
int **isprint**(int c);	Tests c to see if it is a printable ASCII character (including a space)	7-34
int **ispunct**(int c);	Tests c to see if it is an ASCII punctuation character	7-34
int **isspace**(int c);	Tests c to see if it is an ASCII space bar, tab (horizontal or vertical), carriage return, form feed, or new line character	7-34
int **isupper**(int c);	Tests c to see if it is an uppercase ASCII alphabetic character	7-34
int **isxdigit**(int c);	Tests c to see if it is a hexadecimal digit	7-34
char **toascii**(int c);	Masks c into a legal ASCII value	7-58
char **tolower**(int char c);	Converts c to lowercase if it is uppercase	7-59
char **toupper**(int char c);	Converts c to uppercase if it is lowercase	7-59

Note: Functions in ctype.h are expanded inline if the –x option is used.

(c) Inport/outport macros (ioports.h)

Macro	Description	Page
int **inport**(int port, int *ret);	Returns a value from the specified port via the pointer *ret*	7-33
void **outport**(int port, int value);	Writes a value to the specified port and has no return value	7-33

(d) Floating-point math functions (math.h)

Function	Description	Page
double **acos**(double x);	Returns the arc cosine of x	7-21
double **asin**(double x);	Returns the arc sine of x	7-21
double **atan**(double x);	Returns the arc tangent of x	7-23
double **atan2**(double y, double x);	Returns the arc tangent of y/x	7-23
double **ceil**(double x);	Returns the smallest integer ≥ x; expands inline if –x is used	7-26
double **cos**(double x);	Returns the cosine of x	7-27
double **cosh**(double x);	Returns the hyperbolic cosine of x	7-28
double **exp**(double x);	Returns e^x	7-30
double **fabs**(double x);	Returns the absolute value of x	7-30
double **floor**(double x);	Returns the largest integer ≤ x; expands inline if –x is used	7-31
double **fmod**(double x, double y);	Returns the exact floating-point remainder of x/y	7-31
double **frexp**(double value, int *exp);	Returns f and exp such that $.5 \le \lvert f \rvert < 1$ and value is equal to $f \times 2^{exp}$	7-32
double **ldexp**(double x, int exp);	Returns $x \times 2^{exp}$	7-35
double **log**(double x);	Returns the natural logarithm of x	7-36
double **log10**(double x);	Returns the base-10 logarithm of x	7-36
double **modf**(double value, double *ip);	Breaks value into a signed integer and a signed fraction	7-41
double **pow**(double x, double y);	Returns x^y	7-41
double **sin**(double x);	Returns the sine of x	7-45
double **sinh**(double x);	Returns the hyperbolic sine of x	7-45
double **sqrt**(double x);	Returns the nonnegative square root of x	7-46
double **tan**(double x);	Returns the tangent of x	7-56
double **tanh**(double x);	Returns the hyperbolic tangent of x	7-57

(e) Nonlocal jumps macro and function (setjmp.h)

Function or Macro	Description	Page
int **setjmp**(jmp_buf env);	Saves calling environment for use by longjmp; this is a macro	7-44
void **longjmp**(jmp_buf env, int _val);	Uses jmp_buf argument to restore a previously saved environment	7-44

(f) Variable argument macros (stdarg.h)

Macro	Description	Page
type **va_arg**(va_list, type);	Accesses the next argument of type type in a variable-argument list	7-59
void **va_end**(va_list);	Resets the calling mechanism after using va_arg	7-59
void **va_start**(va_list, parmN);	Initializes ap to point to the first operand in the variable-argument list	7-59

(g) General functions (stdlib.h)

Function	Description	Page
void **abort**(void);	Terminates a program abnormally	7-20
int **abs**(int i);	Returns the absolute value of val; expands inline unless –x0 is used	7-20
int **atexit**(void (*fun)(void));	Registers the function pointed to by fun, called without arguments at program termination	7-23
double **atof**(const char *st);	Converts a string to a floating-point value; expands inline if –x is used	7-24
int **atoi**(register const char *st);	Converts a string to an integer	7-24
long **atol**(register const char *st);	Converts a string to a long integer value; expands inline if –x is used	7-24
void ***bsearch**(register const void *key, register const void *base, size_t nmemb, size_t size, int (*compar)(const void *,const void *));	Searches through an array of nmemb objects for the object that key points to	7-25
void ***calloc**(size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes	7-26
div_t **div**(register int numer, register int denom);	Divides numer by denom producing a quotient and a remainder	7-29
void **exit**(int status);	Terminates a program normally	7-30
void **free**(void *packet);	Deallocates memory space allocated by malloc, calloc, or realloc	7-31

(g) General functions (stdlib.h)(Continued)

Function	Description	Page
long **labs**(long i);	Returns the absolute value of i; expands inline unless –x0 is used	7-20
ldiv_t **ldiv**(register long numer, register long denom);	Divides numer by denom	7-29
int **ltoa**(long val, char *buffer);	Converts val to the equivalent string	7-36
void ***malloc**(size_t size);	Allocates memory for an object of size bytes	7-37
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc	7-39
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());	Sorts an array of nmemb members; base points to the first member of the unsorted array, and size specifies the size of each member	7-42
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX	7-43
void ***realloc**(void *packet, size_t size);	Changes the size of an allocated memory space	7-43
void **srand**(unsigned int seed);	Resets the random number generator	7-43
double **strtod**(const char *st, char **endptr);	Converts a string to a floating-point value	7-55
long **strtol**(const char *st, char **endptr, int base);	Converts a string to a long integer	7-55
unsigned long **strtoul**(const char *st, char **endptr, int base);	Converts a string to an unsigned long integer	7-55

(h) String functions (*string.h*)

Function	Description	Page
void ***memchr**(const void *cs, int c, size_t n);	Finds the first occurrence of c in the first n characters of cs; expands inline if –x is used	7-37
int **memcmp**(const void *cs, const void *ct, size_t n);	Compares the first n characters of cs to ct; expands inline if –x is used	7-38
void ***memcpy**(void *s1, const void *s2, register size_t n);	Copies n characters from s1 to s2	7-38
void ***memmove**(void *s1, const void *s2, size_t n);	Moves n characters from s1 to s2	7-38
void ***memset**(void *mem, register int ch, register size_t length);	Copies the value of ch into the first length characters of mem; expands inline if –x is used	7-39
char ***strcat**(char *string1, const char *string2);	Appends string2 to the end of string1	7-46
char ***strchr**(const char *string, int c);	Finds the first occurrence of character c in s; expands inline if –x is used	7-47

(h) String functions (string.h)(Continued)

Function	Description	Page
int **strcmp**(register const char *string1, register const char *s2);	Compares strings and returns one of the following values: < 0 if string1 is less than string2; 0 if string1 is equal to string2; > 0 if string1 is greater than string2. Expands inline if –x is used.	7-47
int **strcoll**(const char *string1, const char *string2);	Compares strings and returns one of the following values: <0 if string1 is less than string2; 0 if string1 is equal to string2; >0 if string1 is greater than string2.	7-47
char ***strcpy**(register char *dest, register const char *src);	Copies string src into dest; expands inline if –x is used	7-48
size_t **strcspn**(register const char *string, const char *chs);	Returns the length of the initial segment of string that is made up entirely of characters that are not in chs	7-48
char ***strerror**(int errno);	Maps the error number in errno to an error message string	7-49
size_t **strlen**(const char *string);	Returns the length of a string	7-50
char ***strncat**(char *dest, const char *src, register size_t n);	Appends up to n characters from src to dest	7-50
int **strncmp**(const char *string1, const char *string2, size_t n);	Compares up to n characters in two strings; expands inline if –x is used	7-51
char ***strncpy**(register char *dest, register const char *src, register size_t n);	Copies up to n characters from src to dest; expands inline if –x is used	7-52
char ***strpbrk**(const char *string, const char *chs);	Locates the first occurrence in string of *any* character from chs	7-53
char ***strrchr**(const char *string, int c);	Finds the last occurrence of character c in string; expands inline if –x is used	7-53
size_t **strspn**(register const char *string, const char *chs);	Returns the length of the initial segment of string, which is entirely made up of characters from chs	7-54
char ***strstr**(register const char *string1, const char *string2);	Finds the first occurrence of string2 in string1	7-54
char ***strtok**(char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2	7-56

(i) Time-support functions (time.h)

Function	Description	Page
char ***asctime**(const struct tm *timeptr);	Converts a time to a string	7-21
clock_t **clock**(void);	Determines the processor time used	7-27
char ***ctime**(const time_t *timer);	Converts calendar time to local time	7-28
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times	7-28
struct tm ***gmtime**(const time_t *timer);	Converts local time to Greenwich Mean Time	7-32
struct tm ***localtime**(const time_t *timer);	Converts time_t value to broken down time	7-35
time_t **mktime**(register struct tm *tptr);	Converts broken down time to a time_t value	7-40
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	Formats a time into a character string	7-49
time_t **time**(time_t *timer);	Returns the current calendar time	7-57

7.4 Description of Run-Time-Support Functions and Macros

This section describes the run-time-support functions and macros. A superscripted number is used in the following descriptions to show exponents. For example, x^y is the equivalent of x to the power y.

abort — *Abort*

Syntax

#include <stdlib.h>

void **abort**(void);

Defined in exit.c in rts.src

Description The abort function usually terminates a program with an error code. The TMS320C2x/C2xx/C5x implementation of the abort function calls the exit function with a value of 0, and is defined as follows:

```
void abort ()
{
   exit(0);
}
```

This makes the abort function equivalent to the exit function.

abs/labs — *Absolute Value*

Syntax

#include <stdlib.h>

int **abs**(int j);
long int **labs**(long int k);

Defined in abs.c in rts.src

Description The C compiler supports two functions that return the absolute value of an integer:

- The abs function returns the absolute value of an integer j.
- The labs function returns the absolute value of a long integer k.

Since int and long int are functionally equivalent types in TMS320C2x/C2xx/C5x C, the abs and labs functions are also functionally equivalent. The abs and labs functions are expanded inline unless the –x0 option is used. For more information, see Section 2.6, *Using Inline Function Expansion*, on page 2-27.

Example

```
int x = –5;
int y = abs (x);     /* abs returns 5 */
```

acos

Arc Cosine

Syntax

#include <math.h>

double **acos**(double x);

Defined in

acos.c in rts.src

Description

The acos function returns the arc cosine of a floating-point argument x, which must be in the range [–1,1]. The return value is an angle in the range [0,π] radians.

Example

```
double realval, radians;

realval = 0.0;
radians = acos(realval); /* acos return π/2 */
return (radians);
```

asctime

Convert Internal Time to String

Syntax

#include <time.h>

char ***asctime**(const struct tm *timeptr);

Defined in

asctime.c in rts.src

Description

The asctime function converts a broken-down time into a string with the following form:

```
Mon Jan 11 11:18:36 1988 \n\0
```

The function returns a pointer to the converted string.

For more information about the functions and types that the time.h header declares and defines, see section 7.2.12, *Time Functions*, on page 7-11.

asin

Arc Sine

Syntax

#include <math.h>

double **asin**(double x);

Defined in

asin.c in rts.src

Description

The asin function returns the arc sine of a floating-point argument x, which must be in the range [–1,1]. The return value is an angle in the range [–π/2,π/2] radians.

Example

```
double realval, radians;

realval = 1.0;
radians = asin(realval);  /* asin returns π/2 */
```

assert — *Insert Diagnostic Information Macro*

Syntax

#include <assert.h>

void **assert**(int expr);

Defined in

assert.h as a macro

Description

The assert macro tests an expression; depending upon the value of the expression, assert either issues a message and aborts execution or continues execution. This macro is useful for debugging.

- If expr is false, the assert macro writes information about the call that failed to the standard output and aborts execution.
- If expr is true, the assert macro does nothing.

The header file that defines the assert macro refers to another macro, NDEBUG. If you have defined NDEBUG as a macro name when the assert.h header is included in the source file, then the assert macro is defined to have no effect.

If NDEBUG is not defined when assert.h is included, the assert macro is defined to test the expression and, if false, write a diagnostic message including the source filename, line number, and test of expression.

The assert macro is defined with the printf function, which is not included in the library. To use assert, you must do one of the following:

- provide your own version of printf
- modify assert to output the message by other means.

Example

In this example, an integer i is divided by another integer j. Since dividing by 0 is an illegal operation, the example uses the assert macro to test j before the division. If j = = 0 assert issues a message and aborts the program.

```
int   i, j;
assert(j);
q = i/j;
```

atan

Polar Arc Tangent

Syntax

#include <math.h>

double **atan**(double x);

Defined in

atan.c in rts.src

Description

The atan function returns the arc tangent of a floating-point argument x. The return value is an angle in the range [–π/2,π/2] radians.

Example

```
double realval, radians;

realval = 1.0;
radians = atan(realval);     /* return value = 0 */
```

atan2

Cartesian Arc Tangent

Syntax

#include <math.h>

double **atan2**(double y, double x);

Defined in

atan.c in rts.src

Description

The atan2 function returns the arc tangent of y/x. The function uses the signs of the arguments to determine the quadrant of the return value. Both arguments cannot be 0. The return value is an angle in the range [–π,π] radians.

Example

```
atan2 (1.0, 1.0)       /* returns π/4 */
atan2 (1.0, –1.0)      /* returns 3π/4 */
atan2 (–1.0, 1.0)      /* returns –π/4 */
atan2 (–1.0, –1.0)     /* returns –3π/4 */
```

atexit

Register Function Callec by Exit()

Syntax

#include <stdlib.h>

void **atexit**(void (*fun)(void));

Defined in

exit.c in rts.src

Description

The atexit function registers the function that is pointed to by fun, to be called without arguments at normal program termination. Up to 32 functions can be registered.

When the program exits through a call to the exit function, a call to abort, or a return from the main function, the functions that were registered are called without arguments in reverse order of their registration.

atof/atoi/atol *Convert String to Number*

Syntax

#include <stdlib.h>

double **atof**(const char *st);
int **atoi**(const char *st);
long int **atol**(const char *st);

Defined in

atof.c and atoi.c, in rts.src

Description

Three functions convert strings to numeric representations:

- The atof function converts a string to a floating-point value. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

- The atoi function converts a string to an integer. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits*

- The atol function converts a string to a long integer. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits*

The *space* is indicated by a space (character), a horizontal or vertical tab, a carriage return, a form feed, or a new line. Following the space is an optional *sign*, and the *digits* that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional sign.

The first character that cannot be part of the number terminates the string.

Since int and long are functionally equivalent in TMS320C2x/C2xx/C5x C, the atoi and atol functions are also functionally equivalent.

The functions do not handle any overflow resulting from the conversion.

Example

```
int i;
double d;
i = atoi ("–3291");  /* i = –3291 */
d = atof ("1.23e–2); /* d = .0123 */
```

bsearch — *Array Search*

Syntax

#include <stdlib.h>

void ***bsearch**(const void *key, const void *base, size_t nmemb,
size_t size, int (*compar)(const void *, const void *));

Defined in

bsearch.c in rts.src

Description

The bsearch function searches through an array of nmemb objects for a member that matches the object that key points to. Argument base points to the first member in the array; size specifies the size (in bytes) of each member.

The contents of the array must be in ascending order. If a match is found, the function returns a pointer to the matching member of the array; if no match is found, the function returns a null pointer (0).

Argument compar points to a function that compares key to the array elements. The comparison function should be declared as:

```
int  cmp(const void *ptr1, const void *ptr2);
```

The cmp function compares the objects that prt1 and ptr2 point to and returns one of the following values:

< 0 if *ptr1 is less than *ptr2
0 if *ptr1 is equal to *ptr2
> 0 if *ptr1 is greater than *ptr2

Example

```
#include <stdlib.h>
#include <stdio.h>

int list [ ] = {1, 3, 4, 6, 8, 9};
int diff (const void *, const void *0;

main( )
{
    int key = 8;
    int p = bsearch (&key, list, 6, 1, idiff);
        /* p points to list[4] */
}
int idiff (const void *il, const void *i2)
{
    return *(int *) i1 – *(int *) i2;
}
```

calloc

Allocate and Clear Memory

Syntax

#include <stdlib.h>

void ***calloc**(size_t num, size_t size);

Defined in

memory.c in rts.src

Description

The calloc function allocates size bytes (size is an unsigned integer or size_t) for each of nmemb objects and returns a pointer to the space. The function initializes the allocated memory to all 0s. If it cannot allocate the memory (that is, if it runs out of memory), it returns a null pointer (0).

The memory that calloc uses is in a special memory pool or heap, defined in an uninitialized named section called .sysmem in memory.c. The constant __SYSTEM_SIZE defines the heap as 1K words. You can change this amount at line time by invoking the linker while the _heap option and specifying the desired size of the heap directlyafter the option. For more information, see section 6.1.4, Dynamic Memory Allocation, on page 6-6.

Example

This example uses the calloc routine to allocate and clear 20 bytes.

```
prt = calloc (20,2);   /*Allocate and clear 20 bytes */
```

ceil

Ceiling

Syntax

#include <math.h>

double **ceil**(double x);

Defined in

ceil.c in rts.src

Description

The ceil function returns a floating-point number that represents the smallest integer greater than or equal to x. The ceil function is inlined if the –x2 option is used.

Example

```
double answer;
answer = ceil(3.1415);   /* answer = 4.0 */
answer = ceil(-3.5);     /* answer = -3.0 */
```

clock — *Processor Time*

Syntax

#include <time.h>

clock_t **clock**(void);

Defined in

clock.c in rts.src

Description

The clock function determines the amount of processor time used. It returns an approximation of the processor time used by a program since the program began running. The time in seconds is the return value divided by the value of the macro CLOCKS_PER_SEC.

If the processor time is not available or cannot be represented, the clock function returns the value of –1.

> **Note: Writing Your Own Clock Function**
>
> The clock function is target-system specific, so you must write your own clock function. You must also define the CLOCKS_PER_SEC macro according to the units of your clock so that the value returned by clock( ) (number of clock ticks) can be divided by CLOCKS_PER_SEC to produce a value in seconds.

For more information about the functions and types that the time.h header declares and defines, see section 7.2.12, Time Function (time.h) on page 7-11.

cos — *Cosine*

Syntax

#include <math.h>

double **cos**(double x);

Defined in

cos.c in rts.src

Description

The cos function returns the cosine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude might produce a result with little or no significance.

Example

```
double radians, cval;          /* cos returns cval */
radians = 3.1415927;
cval = cos(radians);           /* return value = -1.0 */
```

cosh

Hyperbolic Cosine

Syntax

#include <math.h>

double **cosh**(double x);

Defined in cosh.c in rts.src

Description The cosh function returns the hyperbolic cosine of a floating-point number x. A range error occurs (errno is set to the value of EDOM) if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = cosh(x); /* return value = 1.0 */
```

ctime

Calendar Time

Syntax

#include <time.h>

char ***ctime**(const time_t *timer);

Defined in ctime.c in rts.src

Description The ctime function converts the calendar time (pointed to by timer) to local time in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function.

For more information about the functions and types that the time.h header declares and defines, see section 7.2.12, *Time Functions (time.h)*, on page 7-11.

difftime

Time Difference

Syntax

#include <time.h>

double **difftime**(time_t time1, time_t time0);

Defined in difftime.c in rts.src

Description The difftime function calculates the difference between two calendar times, time1 minus time0. The return value is expressed in seconds.

For more information about the functions and types that the time.h header declares and defines, see section 7.2.12, *Time Functions* (time.h), on page 7-11.

div/ldiv — *Division*

Syntax

#include <stdlib.h>

div_t **div**(int numer, denom);
ldiv_t **ldiv**(long numer, denom);

Defined in

div.c in rts.src

Description

Two functions support integer division by returning numer (numerator) divided by denom (denominator). You can use these functions to determine both the quotient and the remainder in a single operation.

- The div function performs integer division. The input arguments are integers; the function returns the quotient and the remainder in a structure of type div_t. The structure is defined as follows:

```
typedef struct
{
   int  quot;          /*  quotient    */
   int  rem;           /*  remainder   */
} div_t;
```

- The ldiv function performs long integer division. The input arguments are long integers; the function returns the quotient and the remainder in a structure of type ldiv_t. The structure is defined as follows:

```
typedef struct
{
   long int  quot;     /*  quotient    */
   long int  rem;      /*  remainder   */
} ldiv_t;
```

If the division produces a remainder, the sign of the quotient is the same as the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. The sign of the remainder is the same as the sign of numer.

Because ints and longs are equivalent types in TMS320C2x/C2xx/C5x C, these functions are also equivalent.

Example

```
int 1 = -10
int j = 3;
div_t result = div (i, j);      /* result.quot == -3 */
          /* result.rem  == -1 */
```

exit

Normal Termination

Syntax

#include <stdlib.h>

void **exit**(int status);

Defined in

exit.c in rts.src

Description

The exit function terminates a program normally. All functions registered by the atexit function are called in reverse order of their registration.

You can modify the exit function to perform application-specific shutdown tasks. The unmodified function simply runs in an infinite loop until the system is reset.

The exit function cannot return to its caller.

The TMS320C2x/C2xx/C5x implementation of the abort function makes it equivalent to the exit function.

exp

Exponential

Syntax

#include <math.h>

double **exp**(double x);

Defined in

exp.c in rts.src

Description

The exp function returns the exponential function of real number x. The return value is the number e^x. A range error occurs if the magnitude of x is too large.

Example

```
double x, y;

x = 2.0;
y = exp(x);         /* y = 7.38905, which is e**2 */
```

fabs

Absolute Value

Syntax

#include <math.h>

double **fabs**(double x);

Defined in

fabs.c in rts.src

Description

The fabs function returns the absolute value of a floating-point number x. The fabs function is expanded inline unless the –x0 option is used.

Example

```
double x, y;

x = –57.5;
y = fabs(x);        /* return value = +57.5 */
```

floor

Floor

Syntax

#include <math.h>

double **floor**(double x);

Defined in

floor.c in rts.src

Description

The floor function returns a floating-point number that represents the largest integer less than or equal to x. The floor function is expanded inline if the –x option is used.

Example

```
double answer;

answer = floor(3.1415);      /* answer = 3.0  */
answer = floor(-3.5);        /* answer = -4.0 */
```

fmod

Floating-Point Remainder

Syntax

#include <math.h>

double **fmod**(double x, double y);

Defined in

fmod.c in rts.src

Description

The fmod function returns the exact floating-point remainder of x divided by y. If y==0, the function returns 0.

Example

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);       /* fmod returns 1.0 */
```

free

Deallocate Memory

Syntax

#include <stdlib.h>

void **free**(void *packet);

Defined in

memory.c in rts.src

Description

The free function deallocates memory space (pointed to by packet) that was previously allocated by a malloc, calloc, or realloc call. This makes the memory space available again. If you attempt to free unallocated space, the function takes no action and returns. For more information, see section 6.1.4, *Dynamic Memory Allocation*, on page 6-6.

Example

This example allocates ten bytes and then frees them.

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

frexp — *Fraction and Exponent*

Syntax

#include <math.h>

double **frexp**(double value, int *exp);

Defined in

frexp30.asm in rts.src

Description

The frexp function breaks a floating-point number into a normalized fraction (f) and the integer power of 2. The function returns f and exp such that 0.5–|f|<1.0 and value ==f x 2^{exp}. The frexp function stores the power is stored in the int pointed to by exp. If value is 0, both parts of the result are 0.

Example

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);

/* after execution, fraction is .75 and exp is 2 */
```

gmtime — *Greenwich Mean Time*

Syntax

#include <time.h>

struct tm ***gmtime**(const time_t *timer);

Defined in

gmtime.c in rts.src

Description

The gmtime function converts a calendar time (pointed to by timer) into Coordinated Universal Time (represented as a broken-down time). The name gmtime has historical significance as Greenwich Mean Time.

For more information about the functions and types that the time.h header declares and defines, see section 7.2.12, *Time Functions* (time.h), on page 7-11.

inport/outport

Get or Send Data To or From a TMS320C2x/C2xx/C5x I/O Port

Syntax

#include <ioports.h>

inport (int port, int *ret);
outport (int port, int value);

Defined in

ioports.asm in rts.src

Description

The following macros are used for accessing the TMS320C2x/C2xx/C5x I/O ports.

- The inport macro reads a value from the specified port and returns the value via the pointer ret.
- The outport macro writes a value to the specified port and has no return value.

These routines are implemented as *macros*, not functions; you cannot use them in expressions. This is an example of the incorrect use of the macro:

```
call (inport (1, &i));       /* Incorrect use of macro*/
```

Instead, the macro must be used as below:

```
inport (1, &i);
call (i);                    /* Correct use           */
```

The port number must be a value between 0 and 15, inclusive. Using any other value as a port number results in undefined behavior.

If you normally use these macros with a constant port number, set _PSWITCH, a constant defined in ioports.h, to 0 (the default). If you normally use these macros with a variable port number, set _PSWITCH to 1. The macros always work, regardless of the value of _PSWITCH.

For additional information on I/O ports, refer to the *TMS320C2x User's Guide,* the *TMS320C2xx User's Guide*, or the *TMS320C5x User's Guide*.

isxxx — *Character Typing*

Syntax

#include <ctype.h>

```
int isalnum(int c);        int islower(int c);
int isalpha(int c);        int isprint(int c);
int isascii(int c);        int ispunct(int c);
int iscntrl(int c);        int isspace(int c);
int isdigit(int c);        int isupper(int c);
int isgraph(int c);        int isxdigit(int c);
```

Defined in

isxxx.c and ctype.c in rts.src
Also defined in ctype.h as macros

Description

These functions test a single argument c to see if it is a particular type of character — alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true , the function returns a nonzero value; if the test is false, the function returns 0. All of the character-typing functions are expanded inline if the –x option is used. The character-typing functions include:

isalnum	identifies alphanumeric ASCII characters (tests for any character for which isalpha or isdigit is true).
isalpha	identifies alphabetic ASCII characters (tests for any character for which islower or isupper is true).
isascii	identifies ASCII characters (characters 0–127).
iscntrl	identifies control characters (ASCII characters 0–31 and 127).
isdigit	identifies numeric characters (0–9).
isgraph	identifies any nonspace character.
islower	identifies lowercase alphabetic ASCII characters.
isprint	identifies printable ASCII characters, including spaces (ASCII characters 32–126).
ispunct	identifies ASCII punctuation characters.
isspace	identifies ASCII spacebar, tab (horizontal or vertical), carriage return, form feed, and newline characters.
isupper	identifies uppercase ASCII alphabetic characters.
isxdigit	identifies hexadecimal digits (0–9, a–f, A–F).

The C compiler also supports a set of macros that perform these same functions. The macros have the same names as the functions but are prefixed with an underscore; for example, _isascii is the macro equivalent of the isascii function. In general, the macros execute more efficiently than the functions.

labs

See abs/labs on page 7-20

ldexp

Multiply by a Power of Two

Syntax

#include <math.h>

double **ldexp**(double x, int exp);

Defined in

ldexp.c in rts.src

Description

The ldexp function multiplies a floating-point number x by 2^{exp} and returns $(x \times 2)^{exp}$. The exponent (exp) can be a negative or a positive value. A range error may occur if the result is too large.

Example

```
double result;

result = ldexp(1.5, 5);          /* result is 48.0 */
result = ldexp(6.0, -3);         /* result is 0.75 */
```

ldiv

See div/ldiv on page 7-29

localtime

Local Time

Syntax

#include <time.h>

struct tm ***localtime**(const time_t *timer);

Defined in

localtime.c in rts.src

Description

The localtime function converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as local time. The function returns a pointer to the converted time.

For more information about the functions and types that the time.h header declares and defines, see section 7.2.12, Time Functions (time.h), on page 7-11.

log

Natural Logarithm

Syntax

#include <math.h>

double **log**(double x);

Defined in

log.c in rts.src

Description

The log function returns the natural logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 2.718282;
y = log(x);         /* Return value = 1.0 */
```

log10

Common Logarithm

Syntax

#include <math.h>

double **log10**(double x);

Defined in

log10.c in rts.src

Description

The log10 function returns the base-10 logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 10.0;
y = log(x);         /* Return value = 1.0 */
```

longjump

See setjmp/longjmp on page 7-44

ltoa

Convert Long Integer to ASCII

Syntax

#include <stdlib.h>

int **ltoa**(long val, char *buffer);

Defined in

ltoa.c in rts.src

Description

The ltoa function converts a long integer val to the equivalent ASCII string and writes it into buffer. If the input number val is negative, a leading minus sign is output. The ltoa function returns the number of characters placed in the buffer.

Example

```
int i;

char s[10];
i = ltoa (−92993L, s);     /* i = 6, s = ”−92993”*/
```

malloc — *Allocate Memory*

Syntax

#include <stdlib.h>

void ***malloc**(size_t size);

Defined in

memory.c in rts.src

Description

The malloc function allocates space for an object of size bytes and returns a pointer to the space. If malloc cannot allocate the packet (that is, if it runs out of memory), it returns a null pointer (0). This function does not modify the memory it allocates.

The memory that malloc uses is in a special memory pool or heap. The constant __SYSTEM_SIZE defines the size of the heap as 1K words. You can change this amount at link time by invoking the linker with the -heap option and specifying the desired size of the heap directly after the option. For more information, see section 6.1.4, *Dynamic Memory Allocation*, on page 6-6.

Example

This example allocates free space for a structure.

```
struct xyz *p;
p = malloc (sizeof (struct xyz));
```

memchr — *Find First Occurrence of Byte*

Syntax

#include <string.h>

void ***memchr**(const void *es, int c, size_t n);

Defined in

memchr.c in rts.src

Description

The memchr function finds the first occurrence of c in the first n characters of the object that es points to. If the character is found, memchr returns a pointer to the located character; otherwise, it returns a null pointer (0).

The memchr function is similar to strchr, except that the object that memchr searches can contain values of 0 and c can be 0. The memchr function is expanded inline when the –x option is used.

memcmp — *Memory Compare*

Syntax

#include <string.h>

int **memcmp**(const void *cs, const void *ct, size_t n);

Defined in

memcmp.c in rts.src

Description

The memcmp function compares the first n characters of the object that ct points to with the object that cs points to. The function returns one of the following values:

< 0 if *cs is less than *ct
0 if *cs is equal to *ct
> 0 if *cs is greater than *ct

The memcmp function is similar to strncmp, except that the objects that memcmp compares can contain values of 0. The memcmp function is expanded inline when the –x option is used.

memcpy — *Memory Block Copy — Nonoverlapping*

Syntax

#include <string.h>

void ***memcpy**(void *s1, const void *s2, size_t n);

Defined in

memcpy.c in rts.src

Description

The memcpy function copies n characters from the object that s2 points to into the object that s1 points to. If you attempt to copy characters of overlapping objects, the function's behavior is undefined. The function returns the value of s1.

The memcpy function is similar to strncpy, except that the objects that memcpy copies can contain values of 0. The memcpy function is expanded inline when the –x option is used.

memmove — *Memory Block Copy — Overlapping*

Syntax

#include <string.h>

void ***memmove**(void *s1, const void *s2, size_t n);

Defined in

memmove.c in rts.src

Description

The memmove function moves n characters from the object that s2 points to into the object that s1 points to; the function returns the value of s1. The memmove function correctly copies characters between overlapping objects.

memset — *Duplicate Value in Memory*

Syntax

#include <string.h>

void ***memset***(void *mem, int ch, size_t length);

Defined in

memset.c in rts.src

Description

The memset function copies the value of ch into the first length characters of the object that mem points to. The function returns the value of mem. The memset function is expanded inline when the –x option is used.

minit — *Reset Dynamic Memory Pool*

Syntax

#include <stdlib.h>

void **minit**(void);

Defined in

memory.c in rts.src

Description

The **minit** function resets all the space that was previously allocated by calls to the malloc, calloc, or realloc functions.

The memory that minit uses is in a special memory pool or heap. The constant __SYSTEM_SIZE defines the size of the heap as 1K words. You can change this amount at link time by invoking the linker with the –heap option specifying the desired size of the heap directly after the option. For more information, see section 6.1.4, *Dynamic Memory Allocation*, on page 6-6.

Note: No Previously Allocated Objects Are Available After minit

Calling the minit function makes *all* the memory space in the heap available again. Any objects that you allocated previously will be lost; do not try to access them.

mktime — *Convert to Calendar Time*

Syntax

#include <time.h>

time_t ***mktime**(struct tm *timeptr);

Defined in

mktime.c in rts.src

Description

The mktime function converts a broken-down time, expressed as local time, into proper calendar time. The timeptr argument points to a structure that holds the broken-down time.

The function ignores the original values of tm_wday and tm_yday and does not restrict the other values in the structure. After successful completion of time conversions, tm_wday and tm_yday are set appropriately and the other components in the structure have values within the restricted ranges. The final value of tm_mday is not sent until tm_mon and tm_year are determined.

The return value is encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value –1.

For more information about the functions and types that the time.h header declares and defines, see subsection 7.2.12, *Time Functions (time.h)*, on page 7-11.

Example

This example determines the day of the week that July 4, 2001 falls on.

```
#include <time.h>
static const char *const wday[] = {
             ”Sunday”, ”Monday”, ”Tuesday”, ”Wednesday”,
             ”Thursday”, ”Friday”, ”Saturday” };

struct tm time_str;

time_str.tm_year  = 2001 – 1900;
time_str.tm_mon   = 7;
time_str.tm_mday  = 4;
time_str.tm_hour  = 0;
time_str.tm_min   = 0;
time_str.tm_sec   = 1;
time_str.tm_isdst = 1;

mktime(&time_str);  /* After calling this function,
                     time_str.tm_wday contains the day of
                     the week for July 4, 2001 */

printf (”result is %s\n”, wday[time_str.tm_wday]);
```

modf — *Signed Integer and Fraction*

Syntax

#include <math.h>

double **modf**(double value, double *iptr);

Defined in

modf.c in rts.src

Description

The modf function breaks a value into a signed integer and a signed fraction. Each of the two parts has the same sign as the input argument. The function returns the fractional part of the value and stores the integer part as a double at the object pointed to by iptr.

Example

```
double value, ipart, fpart;

value = -3.1415;
fpart = modf(value, &ipart);

/* After execution, ipart contains -3.0, */
/*  and fpart contains -0.1415.          */
```

pow — *Raise to a Power*

Syntax

#include <math.h>

double pow**(double x, double y);**

Defined in

pow.c in rts.src

Description

The pow function returns x raised to the power y. A domain error occurs if x = 0 and y ≤ 0, or if x is negative and y is not an integer. A range error occurs if the result is too large to represent.

Example

```
double x, y, z;

x = 2.0;
y = 3.0;
z = pow(x, y);  /* return value = 8.0 */
```

qsort — *Array Sort*

Syntax

#include <stdlib.h>

void **qsort**(void *base, size_t nmemb, size_t size, int (*compar)
(const void *, const void *));

Defined in

qsort.c in rts.src

Description

The qsort function sorts an array of nmemb members. Argument base points to the first member of the unsorted array; argument size specifies the size of each member.

This function sorts the array in ascending order.

Argument compar points to a function that compares key to the array elements. The comparison function should be declared as:

```
int  cmp(ptr1, *ptr2)
   void *ptr1, *ptr2;
```

The cmp function compares the objects that ptr1 and ptr2 point to and returns one of the following values:

< 0 if *ptr1 is less than *ptr2
0 if *ptr1 is equal to *ptr2
> 0 if *ptr1 is greater than *ptr2

Example

In the following example, a short list of integers is sorted with qsort.

```
#include  <stdlib.h>

int list[ ] = {3, 1, 4, 1, 5, 9, 2, 6};
int idiff (const void *, const void *);

main( )
{
   qsort (list, 8, 1, idiff);
   /* after sorting, list[ ]=={ 1, 1, 2, 3, 4, 5, 6, 9} */
}

int idiff (const void *i1, const void *i2)
{
   return *(int *)i1 – *(int *)i2;
```

rand/srand

Random Integer

Syntax

```
#include <stdlib.h>
```

int **rand**(void);
void **srand**(unsigned int seed);

Defined in rand.c in rts.src

Description Two functions work together to provide pseudorandom sequence generation:

- The rand function returns pseudorandom integers in the range 0—RAND_MAX. For the TMS320C2x/C2xx/C5x C compiler, the value of RAND_MAX is 2147483646 (2^{31} –2).
- The srand function sets the value of the seed so that a subsequent call to the rand function produces a new sequence of pseudorandom numbers. The srand function does not return a value.

If you call rand before calling srand, rand generates the same sequence it would produce if you first called srand with a seed value of 1. If you call srand with the same seed value, rand generates the same sequence of numbers.

realloc

Change Heap Size

Syntax

```
#include <stdlib.h>
```

void ***realloc**(void *packet, size_t size);

Defined in memory.c in rts.src

Description The realloc function changes the size of the allocated memory pointed to by packet to the size specified in bytes by size. The contents of the memory space (up to the lesser of the old and new sizes) is not changed.

- If packet is 0, realloc behaves like malloc.
- If packet points to unallocated space, function takes no action and returns.
- If the space cannot be allocated, the original memory space is not changed, and realloc returns 0.
- If size is 0 and packet is not null, realloc frees the space packet points to.

If the entire object must be moved to allocate more space, realloc returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, the function returns a null pointer (0).

The memory that realloc uses is in a special memory pool or heap. The constant __SYSTEM_SIZE defines the size of the heap as 1K words. You can change this amount at link time by invoking the linker with the –heap option specifying the desired size of the heap directly after the option. For more information, see section 6.1.4, *Dynamic Memory Allocation*, on page 6-6.

setjmp/longjmp *Nonlocal Jumps*

Syntax

#include <setjmp.h>

int **setjmp**(jmp_buf env);
void **longjmp**(jmp_buf env, int returnval);

Defined in

setjmp.asm in rts.src

Description

The setjmp.h header defines a type and a macro and declares a function for bypassing the normal function call and return discipline:

- The jmp_buf type is an array type suitable for holding the information needed to restore a calling environment.
- The setjmp macro saves its calling environment in the jmp_buf argument for later use by the longjmp function.

  If the return is from a direct invocation, the setjmp macro returns the value 0. If the return is from a call to the longjmp function, the setjmp macro returns a nonzero value.
- The longjmp function restores the environment that was saved in the jmp_buf argument by the most recent invocation of the setjmp macro. If the setjmp macro was not invoked, or if it terminated execution irregularly, the behavior of longjmp is undefined.

  After longjmp is completed, the program execution continues as if the corresponding invocation of setjmp had just returned returnval. The longjmp function does not cause setjmp to return a value of 0 even if returnval is 0. If returnval is 0, the setjmp macro returns the value 1.

Example

These functions are typically used to effect an immediate return from a deeply nested function call:

```
#include <setjmp.h>

jmp_buf env;

main()
{
   int errcode;

   if ((errcode = setjmp(env)) == 0)
      nest1();
   else
      switch (errcode)
   . . .
}
   . . .
nest42()
{
   if (input() == ERRCODE42)
   /* return to setjmp call in main */
      longjmp (env, ERRCODE42);
   . . .
}
```

sin — *Sine*

Syntax

#include <math.h>

double **sin**(double x);

Defined in sin.c in rts.src

Description The sin function returns the sine of a floating-point number x. The angle x expressed in radians. An argument with a large magnitude can produce a result with little or no significance.

Example

```
double radian, sval;          /* sval is re-
turned by sin */

radian = 3.1415927;
sval = sin(radian);   /* sin returns -1.0        */
```

sinh — *Hyperbolic Sine*

Syntax

#include <math.h>

double **sinh**(double x);

Defined in sinh.c in rts.src

Description The sinh function returns the hyperbolic sine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

Example

```
double x, y;

x = 0.0;
y = sinh(x);      /* return value = 0.0 */
```

sprintf — *Write Stream*

The run-time-support functions supplied with the TMS320C2x/C2xx/C5x C compiler do not include I/O functions such as sprintf. However, since the time function uses sprintf, a minimal version of sprintf() is supplied that performs only the formatting required by time(). See the description of ti_sprintf on page 7-58 for more information.

sqrt — *Square Root*

Syntax

#include <math.h>

double **sqrt**(double x);

Defined in

sqrt.c in rts.src

Description

The sqrt function returns the nonnegative square root of a real number x. A domain error occurs if the argument is negative.

Example

```
double x, y;

x = 100.0;
y = sqrt(x);        /* return value = 10.0 */
```

srand — *See rand/srand on page 7-43*

strcat — *Concatenate Strings*

Syntax

#include <string.h>

char ***strcat**(char *string1, char *string2);

Defined in

strcat.c in rts.src

Description

The strcat function appends a copy of string2 (including a terminating null character) to the end of string1. The initial character of string2 overwrites the null character that originally terminated string1. The function returns the value of string1. The strcat function is expanded inline when the –x option is used string1 must be large enough to contain the entire string.

Example

In the following example, the character strings pointed to by *a, *b, and *c are assigned to point to the strings shown in the comments. In the comments, the notation \0 represents the null character:

```
char *a, *b, *c;
.
.
.
/* a --> "The quick black fox\0"                          */
/* b --> "jumps over \0"                                  */
/* c --> "the lazy dog.\0"                                */

strcat (a,b);
/* a --> "The quick black fox jumps over \0"              */
strcat (a,c);
/* a --> "The quick black fox jumps over the lazy dog.\0"   */
```

strchr — *Find First Occurrence of a Character*

Syntax

#include <string.h>

char ***strchr**(const char *string, char c);

Defined in

strchr.c in rts.src

Description

The strchr function finds the first occurrence of c in string. If strchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0). The strchr function is expanded inline when the –x option is used.

Example

```
char *a = "When zz comes home, the search is on for z's.";
char *b;
char the_z = 'z';

b = strchr(a,the_z);
```

After this example, b points to the first z in zz.

strcmp/strcoll — *String Compare*

Syntax

#include <string.h>

int **strcoll**(const char *string1, const char *string2);
int **strcmp**(const char *string1, const char *string2);

Defined in

strcmp.c in rts.src

Description

The strcmp and strcoll functions compare string2 with string1. The functions are equivalent; both are supported to provide compatibility with ANSI C. The strcmp function is expanded inline when the –x option is used.

The functions return one of the following values:

< 0	if *string1 is less than *string2
0	if *string1 is equal to *string2
> 0	if *string1 is greater than *string2

Example

```
char *stra = "why ask why";
char  *strb = "just do it";
char  *strc = "why ask why";

if (strcmp(stra, strb) > 0)
   {
      /* statements here will be executed      */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here will be executed also */
   }
```

strcpy — *String Copy*

Syntax

#include <string.h>

char ***strcpy**(char *string1, const char *string2);

Defined in

strcpy.c in rts.src

Description

The strcpy function copies string2 (including a terminating null character) into string1. If you attempt to copy strings that overlap, the function's behavior is undefined. The function returns a pointer to string1. The strcpy function is expanded inline when the –x option is used.

Example

In the following example, the strings pointed to by *a and *b are two separate and distinct memory locations. In the comments, the notation \0 represents the null character:

```
char *a = "The quick black fox";
char *b = " jumps over ";

/* a --> "The quick black fox\0"          */
/* b --> " jumps over \0"                 */

strcpy(a,b);

/* a --> " jumps over \0"                 */
/* b --> " jumps over \0"                 */
```

strcspn — *Find Number of Unmatching Characters*

Syntax

#include <string.h>

size_t **strcspn**(const char *string, const char *chs);

Defined in

strcspn.c in rts.src

Description

The strcspn function returns the length of the initial segment of string1, which is made up entirely of characters that are not in string2. If the first character in string1 is in string2, the function returns 0.

Example

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strcspn(stra,strb);    /* length = 0 */
length = strcspn(stra,strc);    /* length = 9 */
```

strerror — *String Error*

Syntax

```
#include <string.h>

char *strerror(int errno);
```

Defined in strerror.c in rts.src

Description The strerror function returns the string “string error”. This function is supplied to provide ANSI compatibility.

strftime — *Format Time*

Syntax

```
#include <time.h>

size_t *strftime(char *out, size_t maxsize, const char *format,
        const struct tm *time);
```

Defined in strftime.c in rts.src

Description The strftime function formats a time (pointed to by time) according to a format string and returns the formatted result in the string out. Up to maxsize characters can be written to out. The format parameter is a string of characters that tells the strftime function how to format the time; the following list shows the valid characters and describes what each character expands to.

%a The abbreviated weekday name (Mon, Tue, . . .)

%A The full weekday name

%b The abbreviated month name (Jan, Feb, . . .)

%B The locale’s full month name

%c The date and time representation

%d The day of the month as a decimal number (0–31)

%H The hour (24-hour clock) as a decimal number (00–23)

%I The hour (12-hour clock) as a decimal number (01–12)

%j The day of the year as a decimal number (001–366)

%m The month as a decimal number (01–12)

%M The minute as a decimal number (00–59)

%p The locale’s equivalent of either A.M. or P.M.

%S The second as a decimal number (00–50)

%U The week number of the year (Sunday is the first day of the week) as a decimal number (00–52)

%x The date representation

%X The time representation

%y The year without century as a decimal number (00–99)

%Y The year with century as a decimal number

%Z The time *zone* name, or by no characters if no time zone exists

For more information about the functions and types that the time.h header declares and defines, see section 7.2.12, page 7-11.

strlen — *Find String Length*

Syntax

#include <string.h>

size_t **strlen**(const char *string);

Defined in strlen.c in rts.src

Description The strlen function returns the length of string. In C, a character string is terminated by the first byte with a value of 0 (a null character). The returned result does not include the terminating null character. The strlen function is expanded inline when the –x option is used.

Example

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strlen(stra);       /* length = 13 */
length = strlen(strb);       /* length = 26 */
length = strlen(strc);       /* length = 7  */
```

strncat — *Concatenate Strings*

Syntax

#include <string.h>

char ***strncat**(char *dest, const char *src, size_t n);

Defined in strncat.c in rts.src

Description The strncat function appends up to n characters of src (including a terminating null character) to dest. The initial character of src overwrites the null character that originally terminated dest; strncat appends a null character to the result. The function returns the value of dest.

Example

In the following example, the character strings pointed to by *a, *b, and *c were assigned the values shown in the comments. In the comments, the notation \0 represents the null character:

```
char *a, *b, *c;
size_t size = 13;
.
.
.
/* a--> "I do not like them,\0"                            */;
/* b--> " Sam I am, \0"                                    */;
/* c--> "I do not like green eggs and ham\0"               */;
strncat (a,b,size);
/* a--> "I do not like them, Sam I am, \0"                 */;
/* b--> " Sam I am, \0"                                    */;
/* c--> "I do not like green eggs and ham\0"               */;
strncat (a,c,size);
/* a--> "I do not like them, Sam I am, I do not like\0"    */;
/* b--> " Sam I am, \0"                                    */;
/* c--> "I do not like green eggs and ham\0"               */;
```

strncmp

Compare Strings

Syntax

#include <string.h>

int **strncmp**(const char *string1, const char *string2, size_t n);

Defined in

strncmp.c in rts.src

Description

The strncmp function compares up to n characters of string2 with string1. The function returns one of the following values:

< 0 if *string1 is less than *string2
0 if *string1 is equal to *string2
> 0 if *string1 is greater than *string2

Example

```
char *stra = "why ask why";
char *strb = "just do it";
char *strc = "why not?";
size_t size = 4;

if (strncmp(stra, strb, size) > 0)
   {
      /* statements here will get executed      */
   }
if (strncmp(stra, strc, size) == 0)
   {
      /* statements here will get executed also */
   }
```

strncpy — *String Copy*

Syntax

#include <string.h>

char ***strncpy**(const char *dest, const char *src, size_t n);

Defined in

strncpy.c in rts.src

Description

The strncpy function copies up to n characters from src into dest. If src is n characters long or longer, the null character that terminates src is not copied. If you attempt to copy characters from overlapping strings, the function's behavior is undefined. If src is shorter than n characters, strncpy appends null characters to dest so that dest contains n characters. The function returns the value of dest.

Example

Note that strb contains a leading space to make it five characters long. Also note that the first five characters of strc are an *I*, a space, the word *am*, and another space, so that after the second execution of strncpy, stra begins with the phrase *I am* followed by two spaces. In the comments, the notation \0 represents the null character.

```
char *stra = "she's the one mother warned you of";
char *strb = " he's";
char *strc = "I am the one father warned you of";
char *strd = "oops";
size_t length = 5;
```

```
strncpy (stra,strb,length);
```

```
/* stra--> " he's the one mother warned you of\0" */;
/* strb--> " he's";\0"                            */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;
```

```
strncpy (stra,strc,length);
```

```
/* stra--> "I am  the one mother warned you of\0" */;
/* strb--> " he's";\0"                            */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;
```

```
strncpy (stra,strd,length);
```

```
/* stra--> "oops\0"                               */;
/* strb--> " he's";\0"                            */;
/* strc--> "I am the one father warned you of\0"  */;
/* strd--> "oops\0"                               */;
```

strpbrk

Find Any Matching Character

Syntax

#include <string.h>

char ***strpbrk**(const char *string, const char *chs);

Defined in

strpbrk.c in rts.src

Description

The strpbrk function locates the first occurrence in string of *any* character in chs. If strpbrk finds a matching character, it returns a pointer to that character; otherwise, it returns a null pointer (0).

Example

```
char *stra = "it wasn't me";
char *strb = "wave";
char *a;

a = strpbrk (stra,strb);
```

After this example, a points to the w in wasn't.

strrchr

Find Last Occurrence of a Character

Syntax

#include <string.h>

char ***strrchr**(const char *string, int c);

Defined in

strrchr.c in rts.src

Description

The strrchr function finds the last occurrence of c in string. If strrchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0). The strrchr function is expanded inline if the –x option is used.

Example

```
char *a = "When zz comes home, the search is on for z's";
char *b;
char the_z = 'z';
```

After this example, *b points to the z near the end of the string.

strspn — *Find Number of Matching Characters*

Syntax

#include <string.h>

size_t ***strspn***(const char *string, const char *chs);

Defined in

strspn.c in rts.src

Description

The strspn function returns the length of the initial segment of string, which is entirely made up of characters in chs. If the first character of string is not in chs, the strspn function returns 0.

Example

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strcspn(stra,strb);    /* length = 3 */
length = strcspn(stra,strc);    /* length = 0 */
```

strstr — *Find Matching String*

Syntax

#include <string.h>

char ***strstr***(const char *string1, const char *string2);

Defined in

strstr.c in rts.src

Description

The strstr function finds the first occurrence of string2 in string1 (excluding the terminating null character). If strstr finds the matching string, it returns a pointer to the located string; if it does not find the string, it returns a null pointer. If string2 points to a string with length 0, strstr returns string1.

Example

```
char *stra = "so what do you want for nothing?";
char *strb = "what";
char *ptr;

ptr = strstr(stra,strb);
```

The pointer *ptr now points to the w in what in the first string.

strtod/strtol/strtoul

Convert String to Numeric Value

Syntax

#include <stdlib.h>

double **strtod**(const char *test, char **endptr);
long int **strtol**(const char *test, char **endptr, int base);
unsigned long int **strtoul**(const char *test, char **endptr, int base);

Defined in

strtod.c in rts.src, strtol.c in rts.src and strtoul.c in rts.src

Description

Three functions convert ASCII strings to numeric values. For each function, argument test points to the original string. Argument endptr points to a pointer; the functions set this pointer to point to the first character after the converted string. The functions that convert to integers also have a third argument, base, which tells the function what base to interpret the string in.

- The strtod function converts a string to a floating-point value. The string must have the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

  The function returns the converted string; if the original string is empty or does not have the correct format, the function returns a 0. If the converted string would cause an overflow, the function returns ± HUGE_VAL; if the converted string would cause an underflow, the function returns 0. If the converted string causes an overflow or an underflow, errno is set to the value of ERANGE.

- The strtol function converts a string to a long integer. The string must have the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

- The strtoul function converts a string to an unsigned long integer. The string must have the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

The space is indicated by a space bar, horizontal or vertical tab, carriage return, form feed, or new line. Following the space is an optional sign and digits that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional sign.

The first unrecognized character terminates the string. The pointer that endptr points to is set to point to this character.

strtok — *Break String Into Token*

Syntax

#include <string.h>

char ***strtok**(char *str1, const char *str2);

Defined in

strtok.c in rts.src

Description

Successive calls to the strtok function break str1 into a series of tokens, each delimited by a character from str2. Each call returns a pointer to the next token. The first call to strtok uses the string str1. Successive calls use a null pointer as the first argument. The value of str2 can change at each invocation. It is important to note that str1 is altered by the strtok function.

Example

After the first invocation of strtok in the example below, the pointer stra points to the string excuse\0 because strtok has inserted a null character where the first space used to be. In the comments, the notation \0 represents the null character.

```
char *stra = ”excuse me while I kiss the sky”;
char *ptr;

ptr = strtok (stra,” ”); /* ptr --> ”excuse\0” */
ptr = strtok (0,” ”);    /* ptr --> ”me\0”     */
ptr = strtok (0,” ”);    /* ptr --> ”while\0”  */
```

tan — *Tangent*

Syntax

#include <math.h>

double **tan**(double x);

Defined in

tan.c in rts.src

Description

The tan function returns the tangent of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double x, y;

x = 3.1415927/4.0;
y = tan(x);           /* return value = 1.0 */
```

tanh — Hyperbolic Tangent

Syntax

#include <math.h>

double **tanh**(double x);

Defined in tanh.c in rts.src

Description The tanh function returns the hyperbolic tangent of a floating-point number x.

Example

```
double x, y;

x = 0.0;
y = tanh(x);        /* return value = 0.0 */
```

time — Time

Syntax

#include <time.h>

time_t **time**(time_t *timer);

Defined in time.c in rts.src

Description The time function determines the current calendar time, represented in seconds since 12:00 A.M., Jan 1, 1900. If the calendar time is not available, the function returns –1. If timer is not a null pointer, the function also assigns the return value to the object that timer points to.

For more information about the functions and types that the time.h header declares and defines, see section 7.2.12, *Time Functions (time.h)*, on page 7-11.

Note: Writing Your Own Time Function

The time function is target-system specific, so you must write your own time function.

ti_sprintf — *Special Version of sprintf*

Syntax

#include <stdlib.h>

int ti_**sprintf** (char *s, const char *format, ...);

Defined in

tsprintf.c in rts.src

Description

The ti_sprintf function is a minimal version of sprintf() that supports only those functions required by time(). Specifically, ti_sprintf supports only the following conversions:

% [**0**] [*digits*] **(s** | **d)**

0 — if present, indicates that the field will be padded with 0s instead of blanks.

digits — if present, indicate the minimum width of the field in characters. If the argument that corresponds to the conversion is smaller than this width, the argument is right justified in the field and the field padded with 0s or blanks (depending on whether 0 is used as above).

s | **d** — specifies the argument's type. An **s** indicates that the argument is of type char *; a **d** indicates that the argument is of type int.

You can alter the ti_sprintf function by extracting the function from rts.src, making changes to the code, and recompiling the run-time-support library. The ti_sprintf function is fully commented to make alterations easier. To extract ti_sprintf.c from the rts.src library, enter the following command at the command line:

```
dspar –x rts.src tsprintf.c
```

toascii — *Convert to ASCII*

Syntax

#include <ctype.h>

int **toascii**(int c);

Defined in

toascii.c in rts.src

Description

The toascii function ensures that c is a valid ASCII character by masking the lower seven bits. There is also an equivalent macro call _toascii.

tolower/toupper — *Convert Case*

Syntax

#include <ctype.h>

int **tolower**(int c);
int **toupper**(int c);

Defined in

tolower.c in rts.src
toupper.c in rts.src

Description

Two functions convert the case of a single alphabetic character c into uppercase or lowercase:

- The tolower function converts an uppercase argument to lowercase. If c is already in lowercase, tolower returns it unchanged.
- The toupper function converts a lowercase argument to uppercase. If c is already in uppercase, toupper returns it unchanged.

The functions have macro equivalents named _tolower and _toupper.

Example

```
tolower ('A')        /* returns 'a' */
tolower ('+')        /* returns '+' */
```

va_arg/va_end/va_start — *Variable-Argument Macros/Functions*

Syntax

#include <stdarg.h>

typedef char ***va_list**;
va_arg(ap, type);
void **va_end**(ap);
void **va_start**(ap, parmN);
va_list *ap

Defined in

stdarg.h as macros

Description

Some functions are called with a varying number of arguments that have varying types. Such a function, called a *variable-argument function*, can use the following macros to step through its argument list at run time. The ap parameter points to an argument in the variable-argument list.

- The va_start macro initializes ap to point to the first argument in an argument list for the variable-argument function. The parmN parameter points to the right-most parameter in the fixed, declared list.
- The va_arg macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies ap so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies ap to point to the next argument in the list). The type parameter is a type name; it is the type of the current argument in the list.

- The va_end macro resets the stack environment after va_start and va_arg are used.

You must call va_start to initialize ap before calling va_arg or va_end.

Example

```
int printf (char *fmt, ...)
{
   va_list ap;
   va_start(ap, fmt);
       .
       .
       .
/* Get next arg, an integer       */
   i = va_arg(ap, int);
/* Get next arg, a string         */
   s = va_arg(ap, char *);
/* Get next arg, a long           */
   l = va_arg(ap, long);
       .
       .
       .
   va_end(ap)       /* Reset      */
}
```

Chapter 8

Library-Build Utility

When using the C compiler, you can compile your code under a number of different configurations and options that are not necessarily compatible with one another. Since it would be cumbersome to include all possible combinations in individual run-time-support libraries, this package includes the source file, rts.src, that contains all run-time-support functions.

You can build your own run-time-support libraries by using the dspmk utility described in this chapter and the archiver described in the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

Topic **Page**

8.1 Invoking the Library-Build Utility 8-2

8.2 Library-Build Utility Options 8-3

8.3 Options Summary 8-4

8.1 Invoking the Library-Build Utility

The dspmk utility runs the shell program on each source file in the archive to compile and/or assemble it. Then, the utility collects all the object files into the object library. All tools must be placed in your PATH. The utility ignores and disables the environment variables TMP, C_OPTION, and C_DIR.

The syntax for invoking the library –build utility is:

dspmk [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

dspmk	is the command that invokes the utility.
options	affect how the library –build utility treats your files. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 8.2 and 8.3.)
src_arch	is the name of a source archive file. For each source archive named, dspmk builds an object library according to the runtime model specified by the command-line options.
–l*obj.lib*	is the optional object library name. If you do not specify a name for the library, dspmk uses the name of the source archive and appends a *.lib* suffix. For each source archive file specified, a corresponding object library file is created. You cannot build an object library from multiple source archive files.

8.2 Library-Build Utility Options

Most of the options that are included on the command line correspond directly to options of the same name used with the compiler, assembler, linker, and shell. The following options apply only to the library-build utility.

−−c Extracts C source files contained in the source archive from the library and leaves them in the current directory after the utility has completed execution.

−−h Uses header files contained in the source archive and leaves them in the current directory after the utility completes execution. Use this option to install the run-time-support header files from the rts.src archive that is shipped with the tools.

−−k Overwrite files. By default, the utility aborts any time it attempts to create an object file when an object file of the same name already exists in the current directory, regardless of whether you specified the name or the utility derived it.

−−q Suppresses header information (quiet).

−−u Does not use the header files contained in the source archive when building the object library. If the desired headers are already in the current directory, there is no reason to reinstall them. This option also gives you flexibility in modifying run-time-support functions to suit your application.

−−v Prints progress information to the screen during execution of the utility. Normally, the utility operates silently (no screen messages).

8.3 Options Summary

The other options you can use with the library-build utility correspond directly to the options that the compiler uses. Table 8–1 lists these options.

Table 8–1. Summary of Options and Their Effects

(a) Options that control the compiler shell

Option	Effect
–g	Enables symbolic debugging
–r*register*	Reserves global register
–v*xx*	Specifies target processor TMS320C*xx* (25, 50, 2xx)

(b) Options that control the parser

Option	Effect
–pk	Makes code K&R compatible
–pw	Suppresses warning messages
–p?	Enables trigraph expansion

(c) Options that control the optimizer

Option	Effect
–o0	Compiles with optimization; register optimization
–o1	Compiles with optimization; + local optimization
–o2 (or –o)	Compiles with optimization; + global optimization
–o3	Compiles with optimization; + file optimization Note that dspmk automatically sets –ol0 and –op0.
–oe	Assumes no calls by interrupts
–ox (equivalent to –x2)	Defines _INLINE + above + invoke optimizer (at –o2 if not specified differently)

(d) Options that control definition-controlled inline function expansion

Option	Effect
–x1	Enables intrinsic function inlining
–x2 (or –x)	Defines _INLINE + above + invoke optimizer (at –o2 if not specified differently)

(e) Options that control the runtime model

Option	Effect
–ma	Assumes aliased variables
–mb	Avoids RPTK for structure moves
–ml	Disables LDPK optimization
–mn	Enables optimization disabled by –g
–ms	Optimizes for space instead of for speed
–mx	Avoids ’C5x silicon bugs

(f) Options that overlook type checking

Options	Effect
–tf	Overlooks relax prototype checking
–tp	Overlooks relax pointer combination checking

(g) Options that control the assembler

Option	Effect
–ap	Enables ’C2x to ’C2xx or ’C5x port switch
–app	Enables ’C2x to ’C2xx port switch and defines .TMS32025 and .TMS3202XX
–as	Keeps labels as symbols

(h) Options that change default file extensions

Options	Effect
–ea[.]	Sets default extension for assembly files
–eo[.]	Sets default extension for object files

Appendix A

Glossary

A

ANSI: *See American National Standards Institute.*

absolute address: An address that is permanently assigned to a memory location.

aliasing: The ability for a single object to be accessed in more than one way, such as when two pointers point to the same object. It can disrupt optimization because any indirect reference could refer to any other object.

allocation: A process in which the linker calculates the final memory addresses of output sections.

American National Standards Institute (ANSI): An organization that establishes standards voluntarily followed by industries.

archive library: A collection of individual files grouped into a single file by the archiver.

archiver: A software program that collects several individual files into a single file called an archive library. With the archiver you can add, delete, extract, or replace members of the archive library.

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.

assignment statement: A statement that initializes a variable with a value.

autoinitialization: The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at runtime: An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke the linker with the –c option. The linker loads the .cinit section of data tables into memory, and variables are initialized at runtime.

B

block: A set of statements that are grouped together within braces and treated as an entity.

.bss section: One of the default COFF sections. You can use the .bss directive to reserve a specified amount of space in the memory map that can later be used for storing data. The .bss section is uninitialized.

byte: Traditionally, a sequence of eight adjacent bits operated upon as a unit. However, the TMS320C2x/C2xx/C5x byte is 16 bits.

Note: TMS320C2x/C2xx/C5x Byte Is 16 Bits

By ANSI C definition, the sizeof operator yields the number of *bytes* required to store an object. ANSI further stipulates that when sizeof is applied to char, the result is 1. Since the TMS320C2x/C2xx/C5x char is 16 bits (to make it separately addressable), a byte is also 16 bits. This yields results you may not expect; for example, sizeof (int) = = 1 (*not* 2). TMS320C2x/C2xx/C5x bytes and words are equivalent (16 bits).

C

C compiler: A software program that translates C source statements into assembly language source statements.

C optimizer: *See optimizer.*

code generator: A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

COFF: *See common object file format.*

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 documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

common object file format (COFF): A system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

constant: A type whose value cannot change

cross-reference listing: An output file created by the assembler that lists the symbols it defined, what line they were defined on, which lines referenced them, and their final values.

D

.data section: One of the default COFF sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

direct call: A function call where one function calls another using the name of the function.

directive: A special-purpose command that controls the actions and functions of a software tool (as opposed to an assembly language instruction, which control the actions of a device).

dynamic memory allocation: A technique used by several functions (such as malloc, calloc, and realloc) to dynamically allocate memory for variables at run time. This is accomplished by defining a large memory pool (heap) and using the functions to allocate memory from the heap.

E

emulator: A hardware development system that duplicates TMS320C2x, TMS320C2xx, or TMS320C5x operation.

entry point: The point in target memory where execution starts.

environment variable: A system symbol that you define and assign to a string. Enviroment variables are often included in batch files, for example, .cshrc.

executable module: A linked object file that can be executed in a target system.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

external symbol: A symbol that is used in the current program module but defined or declared in a different program module.

F

field: For the TMS320C2x, TMS320C2xx, and TMS320C5x, a software-configurable data type whose length can be programmed to be any value in the range of 1–16 bits.

file_leve optimization: A level of optimization where the compiler uses the information that it has about the entire file to optimize your code (as opposed to program_level optimization, where the compiler uses information that it has about the entire program to optimize your code).

function inlining: The process of inserting code for a function at the point of call. This saves the overhead of a function call and allows the optimizer to optimize the function in the context of the surrounding code.

G

global symbol: A symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.

H

hex-conversion utility: A utility that converts COFF object files into one of several standard ASCII hexadecimal formats, suitable for loading into an EPROM programmer.

high-level language debugging: The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

I

indirect call: A function call where one function calls another function by giving the address of the called function.

initialization at load time: An initialization method used by the linker when linking C code. The linker uses this method when you invoke the linker with the –cr option. This method initializes variables at load time instead of run time.

initialized section: A COFF section that contains executable code or data. An initialized section can be built up with the .data, .text, or .sect directive.

integrated preprocessor: The C preprocessor is integrated with the parser, allowing for faster compilation. Stand-alone preprocessing or a preprocessed listing is also available.

interlist utility: A utility that inserts as comments your original C source statements into the assembly language output from the assembler. The C statements are inserted next to the equivalent assembly instructions.

K

K&R C: Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ANSI C compilers should correctly compile and run without modification.

L

label: A symbol that begins in column 1 of an assembly source statement and corresponds to the address of that statement. A label is the only assembler statement that can begin in column 1.

linker: A software program that combines object files to form an object module that can be allocated into system memory and executed by the device.

listing file: An output file created by the assembler that lists source statements, their line numbers, and their effects on the section prgram counter (SPC).

loader: A device that loads an executable module into system memory.

M

macro: A user-defined routine that can be used as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro.

macro expansion: The process of inserting source statements into your code in place of a macro call.

map file: An output file created by the linker that shows the memory configuration, section composition, section allocation, symbol definitions, and the addresses at which the symbols were defined for your program.

memory map: A map of target-system memory space that is partitioned into functional blocks.

O

object file: An assembled or linked file that contains machine-language object code.

object library: An archive library made up of individual object files.

operand: An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

optimizer: A software tool that improves the execution speed and reduces the size of C programs.

options: Command-line parameters that allow you to request additional or specific functions when you invoke a software tool.

output module: A linked, executable object file that is downloaded and executed on a target system.

output section: A final, allocated section in a linked, executable module.

P

parser: A software tool that reads the source file, performs preprocessing functions, checks syntax, and produces an intermediate file that can be used as input for the optimizer or code generator.

partial linking: The linking of a file that will be linked again.

pragma: A preprocessor directive that provides directions to the compiler about how to treat a particular statement.

preprocessor: A software tool that interprets macro definitions, expands macros, interprets header files, interprets conditional compilation, and acts upon preprocessor directives.

R

raw data: Executable code or initialized data in an output section.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

runtime environment: The runtime parameters in which your program must function. These parameters are defiend by the memory and register conventions, stack organization, function call conventions, and system initialization.

runtime-support functions: Standard ANSI functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

runtime-support library: A library file, rts.src, that contains the source for the runtime-support functions as well as for other functions and routines.

S

section: A relocatable block of code or data that will ultimately be contiguous with other sections in the 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.

shell program: A utility that lets you compile, assemble, and optionally link in one step. The shell runs one ore more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

sign extend: To fill the unused MSBs of a value with the value's sign bit.

source file: A file that contains C code or assembly language code that is compiled or assembled to form an object file.

stand-alone preprocessor: A software tool that expands macros, # include files, and conditional compilation as an independent program. It also performs integrated preprocessing, which includes parsing of instructions.

static variable: A variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; the previous values are resumed when the function or program is reentered.

storage class: Any entry in the symbol table that indicates how to access a symbol.

structure: A collection of one or more variables grouped together under a single name.

symbol: A string of alphanumeric characters that represents an address or a value.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

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.

T

target system: The system on which the object code you have developed is executed.

target memory: Physical memory in a TMS320C2x-, TMS320C2xx-, or TMS320C5x-based system into which executable object code is loaded.

.text section: One of the default COFF sections. The .text section that is initialized contains executable code. You can use the .text directive to assemble code into the .text section.

trigraph sequence: A 3-character sequence that has a meaning (as defined by the ISO 646–1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the tragraph ??' is expanded to ^.

U

uninitialized section: A COFF section that reserves space in the memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

unsigned value: A value that is treated as a nonnegative number, regardless of its actual sign.

V

variable: A symbol representing a quantity that can assume any of a set of values.

TOKEN	REFERENCE	SEE (ALSO) . . .
	assembly language	interfacing C with assembly language
	C compiler	compiler
	C language	ANSI C
	C language	interfacing C with assembly language
	C language	K&R
	clist command	interlist utility
	dspac command	parser
	dspac command	preprocessor
	dspcg command	code generator
	dspcl command	compiler
	dsplnk command	linker
	dspmk command	library-build utility
	dspopt command	optimizer
	common object file format	COFF
	diagnostic messages NDEBUG macro	NDEBUG macro
	environment variable	C_DIR
	environment variable	C_OPTION
	environment variable	TMP
	extensions filename	filename extensions
	files intermediate	temporary files
	files, listing	listing files
	files, output	listing files
	files, temporary	temporary files
	intermediate files	temporary files
	Kerrigan & Ritchie C	K&R
	output files	listing files
	parameters function	function parameters
	parameters macro	macros, parameters

TOKEN	REFERENCE	SEE (ALSO) . . .
	parser	preprocessor
	pointer frame	FP register
	pointer frame	frame pointer
	pointer stack	SP register
	pointer stack	stack pointer
	string constants	constants, string
	sprintf function	ti_sprintf function
	system stack	stacks
	time functions, ti_sprintf	ti_sprintf function
	tm structure	broken-down time

Index

A

–a linker option 4-6
–aa shell option 2-19
abort function 7-20
.abs extension 2-15
abs function 7-20
  expanding inline 2-28
absolute compiler limits 5-17
absolute listing, creating 2-19
absolute value 7-20, 7-30
accessing arguments in a function 6-18
accessing local variables in a function 6-18
accumulator 6-10, 6-13
acos function 7-21
–ad shell option 2-19
–ahc shell option 2-19
–ahi shell option 2-19
–al shell option 2-19
aliasing 3-11
allocate memory
  sections 4-11
alternate directories for include files 2-24
ANSI C 1-5
  compatibility with K&R C 5-14 to 5-15
  overlooking type-checking 2-17
  TMS320C2x/C2xx/C5x differs from 5-2 to 5-3
–ap assembler option 2-19
–app assembler option 2-19
append file contents to command line
–ar linker option 4-6
AR0 (FP) 6-4
AR1 (SP) 6-4, 6-32
AR6 5-11, 6-13
AR7 5-11, 6-13
arc cosine 7-21
arc sine 7-21
arc tangent 7-23
archive library
  linking 4-8
archiver 1-3
arguments 6-18
  promotions 2-17
–as shell option 2-19
ASCII conversion functions 7-24
asctime function 7-21, 7-28
asin function 7-21
.asm extension 2-15
asm statement
  and C language 6-22
  described 5-9
  in optimized code 3-10
  masking interrupts 6-26
assembler 1-1, 1-3, 2-38
  options 2-19
assembler control 2-19
assembly language
  See also interfacing C with assembly language
  imbedding in C programs 5-9
  interrupt routines 6-27
  modules 6-19 to 6-21
assembly listing file
  creating 2-19
assert function 7-22
assert.h header 7-5, 7-14
  summary of functions 7-14
atan function 7-23
atan2 function 7-23
atexit function 7-23, 7-30
atof function 7-24
atoi function 7-24
atol function 7-24

–au shell option 2-19
autoinitialization
  at runtime 6-34
  initialization tables 6-33
  of constants 6-32
  of variables 6-7, 6-32
  types of 4-9
–ax shell option 2-19

B

–b interlist option 2-45
–b linker option 4-6
banners
  suppressing 2-13
base 10 logarithm 7-36
bit addressing 6-8
bit fields 5-3, 5-15
block
  memory allocation 4-11
block scope symbols
  maximum number of 5-17
boot.obj 4-8, 4-10, 4-13
broken-down time 7-11, 7-28, 7-40
bsearch function 7-25
.bss section 6-3
  allocating in memory 4-11

C

C compiler 1-3
  See also compiler
  overview 1-5
C entry point 6-25
.c extension 2-15, 2-39
C language
  See also ANSI C; interfacing C with assembly language; K&R
  characteristics 5-2 to 5-3
  integer expression analysis 6-28
  interrupt routine functions 6-25
  interrupt routines 6-26
––c library-build utility option 8-3
–c linker option 4-2, 4-9, 6-4
–c shell option 2-13
  how shell and linker options differ 4-5
C preprocessor 2-22
C source statements and assembly language 2-33
_c_int00
  described 4-10
C_OPTION environment variable 2-20
calendar time 7-11, 7-28, 7-40, 7-57
called function 6-15 to 6-18
calloc function 7-26, 7-31, 7-39
  dynamic memory allocation 6-6
ceil function 7-26
character
  constants 5-15
  conversion functions, summary of 7-14
  string constants 6-8
character sets 5-2
character typing conversion functions 7-5
  isalnum 7-34
  isalpha 7-34
  isascii 7-34
  iscntrl 7-34
  isdigit 7-34
  isgraph 7-34
  islower 7-34
  isprint 7-34
  ispunct 7-34
  isspace 7-34
  isupper 7-34
  isxdigit 7-34
  toascii 7-58
  tolower 7-59
  toupper 7-59
.cinit section 6-3, 6-33
  allocating in memory 4-11
  use during autoinitialization 4-10
.cl extension 2-45
clist command 2-45
  See also interlist utility
CLK_TCK macro 7-11, 7-27
clock function 7-27
clock_t type 7-11
code generator 2-38, 2-43 to 2-44
  invoking 2-43 to 2-44
  options 2-44
code-E error messages 2-35
code-F error messages 2-35
code-I error messages 2-35
code-W error messages 2-35
CODE_SECTION pragma 5-7

COFF 1-3, 1-5, 6-3
command file, linker 4-13
  example 4-13
common logarithm 7-36
common object file format. *See* COFF
compare strings 7-51
compatibility with K&R C 5-14 to 5-15
compiler 1-5
  description 2-1 to 2-46
  error handling 2-35
  invoking 2-4
  limits 5-16 to 5-18
    absolute 5-17
  optimizer 2-38, 3-2 to 3-3
  options 2-6, 2-13 to 2-46
    –@ 2-13
    –c 2-13
    –d 2-13
    –g 2-13, 3-13
    –i 2-13, 2-24
    –k 2-13
    –n 2-13
    –q 2-5, 2-13
    –qq 2-13
    –r 2-13, 5-11
    –s 2-14
    –ss 2-14
    –u 2-14
    –v 2-14
    –z 2-2, 2-14
  overview 1-5 to 1-7, 2-3, 2-38
  running as separate passes 2-38 to 2-46
  sections 4-11
compiling C code 2-2
  with the optimizer 3-2 to 3-3
concatenate strings 7-46, 7-50
.const section 5-12, 6-3, 6-34
  allocating in memory 4-11
  allocating to program memory 6-5
const type qualifier 5-12
constants 5-2
  .const section 5-12
  character 5-2
    escape sequences in 5-15
  floating-point, maximum number of unique 5-18
  string 5-2
    escape sequences in 5-15
    maximum number of unique 5-18
conversions 5-3, 7-5
  C language 5-2
copy file
  –ahc assembler option 2-19
cos function 7-27
cosh function 7-28
cosine 7-27
–cr linker option 4-2, 4-9, 6-7
cross-reference listing
  creating 2-19
ctime function 7-28
ctype.h header 7-5
  summary of functions 7-14

D

–d shell option 2-13
  overriding with –u 2-14
data memory 6-2
.data section 6-3
data types 5-2, 5-4 to 5-5
DATA_SECTION pragma 5-8
__DATE__ 2-22
daylight savings time 7-11
debugging optimized code 3-13
declarations 5-3
dedicated registers 6-12, 6-19
#define
  –d shell option 2-13
defining variables in assembly language 6-23
diagnostic information 7-22
diagnostic messages 7-5
  assert 7-22
  NDEBUG macro. *See* NDEBUG macro
difftime function 7-28
directories
  specifying 2-16
div function 7-29
div_t type 7-10
division 5-3
division and modulus 6-28
_dsp 2-22
dspac command 2-39
  See also parser; preprocessor
dspcg command 2-43
  See also code generator

dspcl command 1-5, 2-4
  See also compiler
dsplnk command 4-2
  See also linker
dspmk command 8-2
  See also library-build utility
dspopt command 2-41
  See also optimizer
dynamic memory allocation
  described 6-6

E

–e linker option 4-6
–ea shell option 2-16
EDOM macro 7-6
entry points
  system reset 6-25
enumerator list
  trailing comma 5-15
environment variable
  See also C_DIR; C_OPTION; TMP
  C_DIR 2-23
  C_OPTION 2-20
  TMP 2-21
–eo shell option 2-16
EPROM programmer 1-4
ERANGE macro 7-6
errno.h header 7-6
error
  creating listing 2-36
  message macro 7-14
  messages from the preprocessor 2-22
#error directive 2-26
error handling 2-35 to 2-37, 5-14
  using error options 2-37
error message macros 7-14
  assert 7-22
error messages
  code-E 2-35
  code-F 2-35
  code-I 2-35
  code-W 2-35
  general 2-35
error options 2-37
error reporting 7-6
errors treated as warnings 2-36
escape sequences 5-2, 5-15
exit function 7-20, 7-23, 7-30
exp function 7-30
exponential math function 7-9, 7-30
expression analysis
  floating-point 6-30
  integers 6-28
expression registers 6-13
expressions 5-3
extensions
  abs 2-15
  asm 2-15
  c 2-15
  filename. *See* filename extensions
  nfo 3-5
  obj 2-15
  s 2-15
  specifying 2-15
external declarations 5-14
external variables 6-7

F

–f linker option 4-6
–fa shell option 2-15
fabs function 7-30
  expanding inline 2-28
fatal errors 2-35
  increasing the threshold of 2-36
–fc shell option 2-15
field manipulation 6-8
file
  copy 2-19
  include 2-19
__FILE__ 2-22
file-level optimization 3-4
filename
  extension specification 2-15
  specifications
    maximum length 5-17
  specifying 2-15
files
  intermediate. *See* temporary files
  listing. *See* listing files
  output. *See* listing files
  temporary. *See* temporary files

float.h header 7-6

floating-point
- expression analysis 6-30
- math functions 7-9
  - *acos 7-21*
  - *asin 7-21*
  - *atan 7-23*
  - *atan2, 7-23*
  - *ceil 7-26*
  - *cos 7-27*
  - *cosh 7-28*
  - *exp 7-30*
  - *fabs 7-30*
  - *floor 7-31*
  - *fmod 7-31*
  - *frexp 7-32*
  - *ldexp 7-35*
  - *log 7-36*
  - *log10, 7-36*
  - *modf 7-41*
  - *pow 7-41*
  - *sinh 7-45*
  - *sqrt 7-46*
  - *tan 7-56*
  - *tanh 7-57*
- remainder 7-31
- summary of functions 7-15 to 7-17

floor function 7-31

fmod function 7-31

–fo shell option 2-15

format time 7-49

FP register 6-4

–fr shell option 2-16

frame pointer 6-4, 6-11 to 6-12

free function 7-31

frexp function 7-32

–fs shell option 2-16

–ft shell option 2-16

FUNC_EXT_CALLED pragma
- described 5-8
- use with –pm option 3-8

function
- alphabetic reference 7-20
- call 6-15
  - *conventions 6-14 to 6-18*
  - *using the stack 6-4*
- general utility 7-16
- inlining 2-27 to 2-32
- parameters
  - *maximum number of 5-17*
- prototype
  - *overlooking type-checking 2-17*

G

–g linker option 4-6

–g shell option 2-13, 3-13

general utility functions 7-10
- abort 7-20
- abs 7-20
- atexit 7-23
- atof 7-24
- atoi 7-24
- atol 7-24
- bsearch 7-25
- calloc 7-26
- div 7-29
- exit 7-30
- free 7-31
- labs 7-20
- ldiv 7-29
- ltoa 7-36
- malloc 7-37
- minit 7-39
- qsort 7-42
- rand 7-43
- realloc 7-43, 7-45
- srand 7-43
- strtod 7-55
- strtol 7-55
- strtoul 7-55
- ti_sprintf 7-58

global symbols
- maximum number of 5-17

global variables 5-12, 6-7
- reserved space 6-3

gmtime function 7-32

gregorian time 7-11

H

––h library-build utility option 8-3
–h linker option 4-6
header files 7-4 to 7-12
  assert.h header 7-5
  ctype.h header 7-5
  errno.h header 7-6
  float.h header 7-6
  limits.h header 7-6
  math.h header 7-9
  setjmp.h header 7-9
  stdarg.h header 7-9
  stddef.h header 7-10
  stdlib.h header 7-10
  string.h header 7-11
  time.h header 7-11
heap
  described 6-6
  reserved space 6-3
–heap linker option 4-6, 7-37
hex conversion utility 1-4
HUGE_VAL 7-9
hyperbolic
  cosine 7-28
  math function 7-9
  sine 7-45
  tangent 7-57

I

–i linker option 4-6
–i shell option 2-13, 2-23, 2-24
  maximum number of 2-24
identifiers 5-2
#if maximum nesting 5-17
.if extension 2-39
implementation errors 2-35
implementation-defined behavior 5-2 to 5-3
#include
  files 2-22, 2-23
    search paths 2-23
  –i shell option 2-13
  maximum file nesting 5-17
  maximum search paths 5-17
include files 2-19
#include preprocessor directive 7-4
INDX register 6-12
  shadow register capability 6-27
initialization
  at load time 6-35
  of variables 6-7
  types 4-9
initialization tables 6-33
initialized sections 6-3
  allocating in memory 4-11
initializers
  local maximum number 5-17
initializing global variables 5-12
initializing static variables 5-12
_INLINE 2-22
  preprocessor symbol 2-31
inline
  declaring functions as 2-28
  expansion 2-27 to 2-32
  keyword 2-28
inline assembly construct (asm) 6-22
inline assembly language 6-22
inlining
  automatic expansion 3-12
integer division 7-29
integer expression analysis 6-28
  division and modulus 6-28
  overflow and underflow 6-28
interfacing C and assembly language 6-19
  asm statement 6-22
  assembly language modules 6-19 to 6-21
interlist utility 1-3, 1-6, 2-33
  invoking 2-14, 2-45
  options 2-45
    –b 2-45
    –q 2-45
    –r 2-45
  used with the optimizer 3-13
intermediate files
  See also temporary files
  code generator 2-43
  optimizer 2-42
  parser 2-39
interrupt handling 6-25 to 6-27
intrinsic operators 2-27, 2-28
inverse tangent of y/x 7-23

invoking the
- C compiler 2-4
- C compiler tools individually 2-38
- code generator 2-43
- interlist utility 2-33, 2-45
- library-build utility 8-2
- linker 4-2
- optimizer 2-41
- parser 2-39

ioport keyword 5-13

isalnum function 7-34

isalpha function 7-34

isascii function 7-34

iscntrl function 7-34

isdigit function 7-34

isgraph function 7-34

islower function 7-34

isprint function 7-34

ispunct function 7-34

isspace function 7-34

isupper function 7-34

isxdigit function 7-34

isxxx function 7-5, 7-34

J

jump function 7-16

jump macro 7-16

K

––k library-build utility option 8-3

–k shell option 2-13

Kernighan & Ritchie C. *See* K&R

L

–l library-build utility option 8-2

–l linker option 4-2, 4-8

label, retaining 2-19

labs function 7-20
- expanding inline 2-28

ldexp function 7-35

ldiv function 7-29

ldiv_t type 7-10

libraries 7-2

library-build utility 1-3, 1-6, 8-1 to 8-6
- optional object library 8-2
- options 8-3

limits
- absolute compiler 5-17
- compiler 5-16 to 5-18
- floating-point types 7-6
- integer types 7-6

limits.h header 7-6

#line directive 2-25

__LINE__ 2-22

linker 1-3, 2-39
- command file 4-13 to 4-14
  - *example 4-13*
- disabling 4-5
- invoking individually 4-2
- options 4-6 to 4-7
  - *–a 4-6*
  - *–ar 4-6*
  - *–b 4-6*
  - *–e 4-6*
  - *–f 4-6*
  - *–g 4-6*
  - *–h 4-6*
  - *–heap 4-6*
  - *–i 4-6*
  - *–l 4-6*
  - *–m 4-6*
  - *–n 4-6*
  - *–o 4-7*
  - *–q 4-7*
  - *–r 4-7*
  - *–s 4-7*
  - *–stack 4-7*
  - *–u 4-7*
  - *–v0 4-7*
  - *–v1 4-7*
  - *–v2 4-7*
  - *–w 4-7*
  - *–x 4-7*
- suppressing 2-13

linking
- C code 4-1 to 4-12
- individually 4-2
- object library 7-2
- with run-time-support libraries 4-8
- with the shell program 4-4

listing file 2-25
  assembly language
    –k shell option 2-13
    register use (–mr option) 2-18
  creating cross-reference 2-19
load time initialization 6-35
loader 5-12
local initializers maximum number 5-17
local time 7-11, 7-28, 7-40
local variable pointer 6-11 to 6-12, 6-18
local variables 6-18
localtime function 7-35
log function 7-36
log10 function 7-36
longjmp function 7-44
ltoa function 7-36

M

–m linker option 4-6
–ma runtime-model option 2-18
macros
  alphabetic reference 7-20
  definitions 2-22 to 2-23
  expansions 2-22 to 2-23
  maximum defined with –d 5-17
  maximum nesting level 5-17
  parameters
    maximum number 5-17
malloc function 7-31, 7-37, 7-39
  dynamic memory allocation 6-6
math.h header 7-9
  summary of functions 7-15 to 7-17
–mb runtime-model option 2-18
memchr function 7-37
memcmp function 7-38
memcpy function 7-38
memmove function 7-38
memory
  data 6-2
  program 6-2
memory management functions
  calloc 7-26
  free 7-31
  malloc 7-37
  minit 7-39
  realloc 7-43, 7-45
memory model 6-2 to 6-8
  allocating variables 6-7
  autoinitialization at run time 6-7
  dynamic memory allocation 6-6
  field manipulation 6-8
  initialization at load time 6-7
  sections 6-3
  stack 6-4
  structure packing 6-8
memory pool 7-37
  reserved space 6-3
memset function 7-39
minit function 7-39
mktime function 7-40
–ml run-time-model option 2-18
–mn run-time-model option 2-18, 3-13
modf function 7-41
modifying compiler output 6-24
modulus 6-28
–mr runtime-model option 2-18
–ms runtime-model option 2-18
multibyte characters 5-2
–mx runtime-model option 2-18

N

–n linker option 4-6
–n shell option 2-13
natural logarithm 7-36
NDEBUG macro 7-5, 7-22
nesting maximum number of
  #include files 5-17
  conditional inclusion (#if) 5-17
  declarations 5-17
  macro levels 5-17
.nfo extension 3-5
nonlocal jump function and macro 7-9
  summary of 7-16
nonlocal jumps 7-44
NULL macro 7-10

O

–o linker option 4-7

–o shell option 3-2

.obj extension 2-15

object library
  linking code with 7-2

–oe shell option 3-11

offsetof macro 7-10

–oi shell option 3-12

–ol shell option 3-4

–on shell option 3-5

–op shell option 3-6 to 3-8

optimization
  algebraic reordering 3-18
  alias disambiguation 3-18
  autoincrement addressing 3-15
  branch optimizations 3-20
  calls 3-16
  common subexpression elimination 3-18
  constant folding 3-18
  control flow simplification 3-20
  copy propagation 3-18
  cost-based register allocation 3-15
  delayed branches 3-16
  inline function expansion 3-21
  loop induction variable optimizations 3-21
  loop invariant code motion 3-21
  loop rotation 3-21
  redundant assignment elimination 3-18
  repeat blocks 3-15
  strength reduction 3-21
  symbolic simplification 3-18

optimizations
  controlling the level of 3-6
  file-level 3-4
  information file options 3-5
  levels 3-2
  list of 3-14 to 3-22
  program-level 3-6

optimized code
  debugging 3-13

optimizer 1-3, 2-41 to 2-43
  and interrupts 3-11
  invoking 2-41
  invoking with shell options 3-2
  options 2-42
  parser output 2-42
  special considerations 3-10
    aliasing 3-11
    volatile keyword 3-10
  use with debugger 2-18

options 2-6 to 2-19
  assembler 2-19
  code generator 2-44
  conventions 2-6
  general 2-13 to 2-46
  interlist utility 2-45
  linker 4-6 to 4-7
  optimizer 2-42
  parser 2-40
  run-time-model 2-18 to 2-20
  summary table 2-7

output files. *See* listing files

overflow
  arithmetic 6-28
  runtime stack 6-32

P

–p? parser option 2-25

packing structures 6-8

parameters
  function. *See* function parameters
  macros. *See* macros, parameters

parser 2-38, 2-39 to 2-40
  See also preprocessor
  options 2-39, 2-40

parsing in two passes 2-41

–pe parser option 2-36

–pk parser option 5-15

–pl parser option 2-25

–pm shell option 3-6

–po parser option 2-41

pointer
  frame. *See* FP register; frame pointer
  stack. *See* SP register; stack pointer

pointer combinations 5-14

port ’C2x assembly code to ’C2xx 2-19

port ’C2x assembly code to ’C2xx or ’C5x 2-19

port variables
ioport keyword 5-13
pow function 7-41
power 7-41
#pragma directive 5-3
pragma directives
CODE_SECTION 5-7
DATA_SECTION 5-8
FUNC_EXT_CALLED 5-8
predefined names 2-22 to 2-23
–ad assembler option 2-19
DATE 2-22
_dsp 2-22
FILE 2-22
_INLINE 2-22
LINE 2-22
TIME 2-22
_TMS320C25 2-22
_TMS320C2xx 2-22
_TMS320C50 2-22
preinitialized 5-12
preprocessed listing file 2-25
preprocessor 2-22 to 2-26
#error directive 2-26
#warn directive 2-26
error messages 2-22
_INLINE symbol 2-31
symbols 2-22
preprocessor directives 2-22
C language 5-3
trailing tokens 5-15
processor time 7-27
program memory 6-2
program termination functions
abort (exit) 7-20
atexit 7-23
exit 7-30
program-level optimization
controlling 3-6
performing 3-6
prototype functions 2-17
nesting of declarations
maximum number of 5-17
pseudorandom 7-43
ptrdiff_t 5-2
ptrdiff_t type 7-10
–pw parser option 2-36

Q

––q library-build utility option 8-3
–q interlist option 2-45
–q linker option 4-7
–q shell option 2-5, 2-13
–qq shell option 2-13
qsort function 7-42

R

–r interlist option 2-45
–r linker option 4-7
–r shell option 2-13, 5-11
rand function 7-43
RAND_MAX macro 7-10
realloc function 6-6, 7-31, 7-39, 7-43, 7-45
recoverable errors 2-35
register conventions 6-9 to 6-13
register variables 5-6
register storage class 5-3
register variables 5-6, 6-12, 6-13
C language 5-6
global 5-10
used with optimizer 6-13
used without optimizer 6-12
registers
accumulator 6-10, 6-13
during function calls 6-15 to 6-18
frame pointer (FP) 6-4, 6-11 to 6-12
INDX 6-12
local variable pointer (LVP) 6-11 to 6-12, 6-18
stack pointer (SP) 6-4, 6-11 to 6-12
use
conventions 6-10
information (–mr option) 2-18
related documentation, vii
RETD instruction 6-18
return values 6-13
RPTK instruction 2-18
rts.src 7-10
rts25.lib 1-3
rts2xx.lib 1-3
rts50.lib 1-3

run-time environment 6-1 to 6-36
  defining variables in assembly language 6-23
  floating-point expression analysis 6-30
  function call conventions 6-14 to 6-18
  inline assembly language 6-22
  integer expression analysis 6-28
  interfacing C with assembly language 6-19 to 6-24
  interrupt handling 6-25 to 6-27
  memory model
    allocating variables 6-7
    dynamic memory allocation 6-6
    field manipulation 6-8
    RAM model 6-7
    ROM model 6-7
    sections 6-3
    structure packing 6-8
  modifying compiler output 6-24
  register conventions 6-9 to 6-13
  stack 6-4
  system initialization 6-31 to 6-36
run time initialization 6-34
run-time-model options 2-18 to 2-20
  –ma 2-18
  –mb 2-18
  –ml 2-18
  –mn 2-18, 3-13
  –mr 2-18
  –ms 2-18
  –mx 2-18
run-time-support
  functions
    introduction 7-1
    summary 7-13
  libraries 7-2, 8-1
    linking C code 4-2, 4-8
    rts.src 8-1
  macros summary 7-13

S

.s extension 2-15
–s linker option 4-7
–s shell option 2-14, 2-33
searches 7-25
sections 6-3
  allocating memory 4-11
  .bss 6-3
  .cinit 6-4, 6-33
  created by the compiler 4-11
  .data 6-3
  .stack 6-3
  .sysmem 6-3
  .text 6-3
setjmp function 7-44
setjmp.h header 7-9
  summary of functions and macros 7-16
shell program 1-3, 2-4 to 2-12
  overview 2-2
  summary of options 2-6
shift 5-3
sinh function 7-45
size_t 5-2
size_t type 7-10
software development tools 1-2 to 1-4
sorts 7-42
source file
  extensions 2-15
source line
  maximum length 5-17
SP register 6-4
sprintf function. *See* ti_sprintf function
sqrt function 7-46
square root 7-46
srand function 7-43
–ss shell option 2-14
–ss shell option 3-13
stack 6-4, 6-32
  overflow of runtime stack 6-32
  reserved space 6-3
–stack linker option 4-7
stack management 6-4
stack pointer 6-4, 6-11 to 6-12, 6-32
.stack section 6-3
  allocating in memory 4-11
__STACK_SIZE constant 6-5
static inline functions 2-30
static variables 5-12, 6-7
  reserved space 6-3
status register fields 6-11
stdarg.h header 7-9
  summary of macros 7-16

stddef.h header 7-10
stdlib.h header 7-10
  summary of functions 7-16
strcat function 7-46
strchr function 7-47
strcmp function 7-47
strcoll function 7-47
strcpy function 7-48
strcspn function 7-48
strerror function 7-49
strftime function 7-49
string constants. *See* constants, string
string copy 7-52
string functions 7-11, 7-17
  memchr 7-37
  memcmp 7-38
  memcpy 7-38
  memmove 7-38
  memset 7-39
  strcat 7-46
  strchr 7-47
  strcmp 7-47
  strcoll 7-47
  strcpy 7-48
  strcspn 7-48
  strerror 7-49
  strlen 7-50
  strncat 7-50
  strncmp 7-51
  strncpy 7-52
  strpbrk 7-53
  strrchr 7-53
  strspn 7-54
  strstr 7-54
  strtok 7-56
string.h header 7-11
  summary of functions 7-17
strlen function 7-50
strncat function 7-50
strncmp function 7-51
strncpy function 7-52
strpbrk function 7-53
strrchr function 7-53
strspn function 7-54
strstr function 7-54
strtod function 7-55
strtok function 7-56
strtol function 7-55
strtoul function 7-55
structure members 5-3
structure packing 6-8
structures
  nesting of declarations
    maximum number of 5-17
STYP_CPY flag 4-10
suppress
  all output except error messages 2-13
  warning messages 2-36
.switch section 6-3
  allocating in memory 4-11
symbol table
  creating labels 2-19
symbolic cross-reference 2-19
symbolic debugging 2-45
  directives 2-13
symbols
  block scope
    maximum visible at any point 5-17
  defined by the assembler 2-19
  global
    maximum number of 5-17
  undefined by the assembler 2-19
.sysmem section 6-3
  allocating in memory 4-11
__SYSMEM_SIZE 6-6
  memory management 7-10
system constraints
  __STACK_SIZE 6-5
  __SYSMEM_SIZE 6-6
system initialization 6-31 to 6-36
  autoinitialization 6-32
  initialization tables 6-33
  stack 6-32
system stack 6-4
  See also stacks

T

tan function 7-56
tangent 7-56
tanh function 7-57
target processor 2-14

temporary files
  code generator 2-43
  optimizer 2-42
  parser 2-39
tentative definition 5-15
.text section 6-3
  allocating in memory 4-11
–tf option, shell 2-17
The C Programming Language 5-14 to 5-15
ti_sprintf function 7-58
time function 7-57
time functions 7-11, 7-14
  asctime 7-21
  clock 7-27
  ctime 7-28
  difftime 7-28
  gmtime 7-32
  localtime 7-35
  mktime 7-40
  strftime 7-49
  summary of 7-19
  ti_sprintf. *See* ti_sprintf function
  time 7-57
time.h header 7-11, 7-14
  summary of functions 7-19
__TIME__ 2-22
time_t type 7-11
tm structure 7-11
  See also broken-down time
TMP environment variable 2-21
_TMS320C25, 2-22
TMS320C2x/C2xx/C5x C language, compatibility with ANSI C language 5-14 to 5-15
_TMS320C2xx 2-22
_TMS320C50, 2-22
toascii function 7-58
tokens 7-56
tolower function 7-59
toupper function 7-59
trailing comma, enumerator list 5-15
trailing tokens, preprocessor directives 5-15
translation phases 2-25
trigonometric math function 7-9
trigraph sequences 2-25
type-checking, overlooking 2-17

U

––u library-build utility option 8-3
–u linker option 4-7
–u shell option 2-14
undefine predefined names, –au assembler option 2-19
underflow 6-28
uninitialized sections 6-3
  allocating in memory 4-11
  .bss 6-3
unions, nesting of declarations, maximum number of 5-17

V

––v library-build utility option 8-3
–v shell option 2-14
–v0 linker option 4-7
–v1 linker option 4-7
–v2 linker option 4-7
va_arg function 7-59
va_end function 7-59
va_start function 7-59
variable allocation 6-7
variable argument functions and macros 7-9
  va_arg 7-59
  va_end 7-59
  va_start 7-59
variable argument macros, summary of 7-16
variable argument function 7-59
variables
  register
    global 5-10
volatile 3-10

W

–w linker option 4-7
#warn directive 2-26
warning messages 2-35, 5-14
  suppressing 2-36
wildcards
  use 2-15

X

–x linker option 4-7

Z

–z shell option 2-2, 2-4, 2-14
overriding with –c option 4-5
overriding with –n option 2-13