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

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

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

TI warrants performance of its semiconductor products and related software 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, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales offices.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.

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
- Preprocessor
- 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 TMS320C2x/C2xx/C5x optimizing 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, describes C based on the ANSI C standard. Use the Kernighan and Ritchie book as a supplement to this manual.

Before you use this book, 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, 3, 4, 5, and 6, 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 Appendices A and B, provides supplementary information on TMS320C2x/C2xx/C5x specific optimizations, and 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 program listing:

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

- 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** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Syntax that will be entered on a command line is centered in a bounded box. Syntax that will be used in a text file is left justified in an unbounded box. 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 don't enter the brackets themselves. Here's 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.

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

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, describes ANSI C. You can use it as a reference.

You may find these documents helpful as well:

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

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

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

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

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, please identify the book by its title and literature number.

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.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Request more information about Texas Instruments digital signal processing (DSP) products	Call the DSP hotline: **(713) 274–2320** Write to: Texas Instruments Incorporated Market Communications Mgr, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320** or fax your request: **(713) 274–2324**
Report mistakes or make comments about this, or any other TI documentation	Send your comments to **comments@books.sc.ti.com**
Please mention the full title of the book and the date of publication (from the spine and/or front cover) in your correspondence.	Texas Instruments Incorporated Technical Publications Mgr, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

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 SunWorkstation are trademarks of Sun Microsystems, Inc.

XDS is a trademark of Texas Instruments Incorporated.

VMS is a trademark of Digital Equipment Corp.

Contents

Figures

Tables

Examples

Notes

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 fully supported by a complete set of code generation tools including an optimizing C compiler, an assembler, a linker, an archiver, a software simulator, a full-speed emulator, and a software development board.

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.

Texas Instruments provides a hotline to assist you with technical questions about the TMS320 family products and development tools. The phone number is (713) 274-2320.

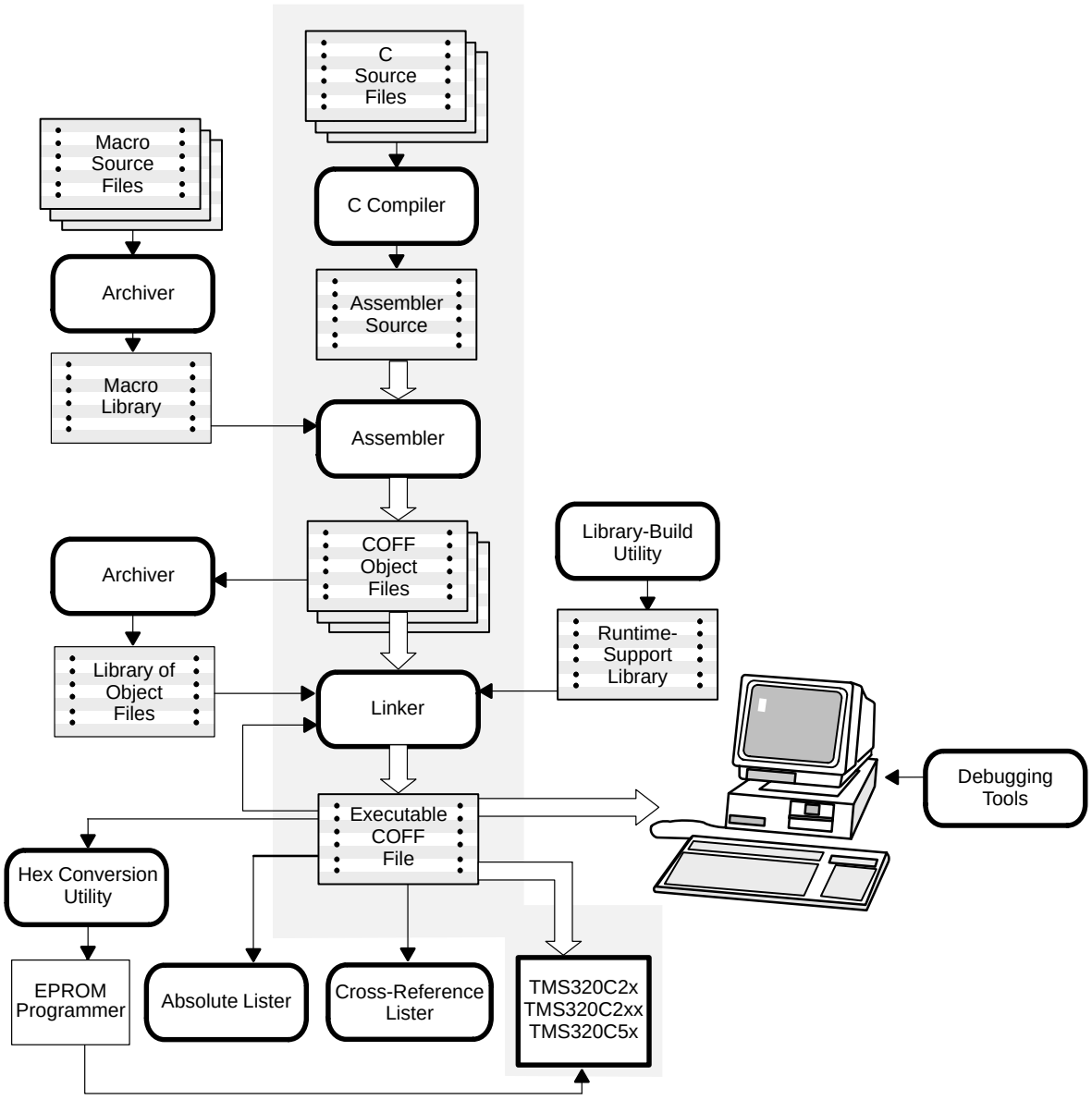
These are the topics included in this introductory chapter:

Topic **Page**

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; the other portions are optional.

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 interlist utility shows the assembly language generated for each source statement. Chapter 2 describes how to invoke and operate the compiler, the shell, the optiomizer, and the interlist utility.

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

- You can use the **library-build utility** to build your own customized runtime-support library. Standard runtime-support library functions are provided as source code located rts.src. See Chapter 6.

- The **archiver** allows you to collect a group of files into a single archive file, called a *library.* Additionally, the archiver allows you to 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 runtime-support functions and compiler-utility functions for the TMS320C2x.
  - **rts50.lib** contains ANSI standard runtime-support functions and compiler-utility functions for the TMS320C5x.
  - **rts2xx.lib** contains ANSI standard runtime-support functions and compiler-utility functions for the TMS320C2xx.

- 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 **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** generates a file that can be reassembled to produce a listing of the absolute addresses of an object file.

- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definition, and their references in the linked source files.

The main purpose of this development process is to produce a module that can be executed in a **TMS320C2x/C2xx/C5x target system**. 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 TMS320C2x/C2xx/C5x 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 features 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 Kernighan and Ritchie's *The C Programming Language* (second edition). The ANSI standard includes recent extensions to C that are now standard features of the language. These extensions provide maximum portability and increased capability.

- **ANSI Standard Runtime Support**

  The compiler package comes with a complete runtime library for each device: 'C2x, 'C2xx and 'C5x. All library functions conform to the ANSI C library standard. The libraries include functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, plus exponential and hyperbolic functions. Functions for I/O and signal handling are not included because these are target-system specific. For more information, refer to Chapter 5.

- **Optimization**

  The compiler uses a sophisticated 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 features specific to the TMS320C2x/C2xx/C5x architecture. For more information about the C compiler's optimization techniques, refer to Section 2.3 on page 2-25 and to Appendix A.

- **Assembly Source Output**

  The compiler generates assembly language source that is easily inspected, 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 provides rich support for source-level debugging.

- **Compiler Shell Program**

  The compiler package includes a shell program that enables you to compile and link programs in a single step. For more information, refer to Section 2.1 on page 2-2.

- **Source Interlist Utility**

  The compiler package includes a utility that interlists your original C source statements into the assembly language output of the compiler. This utility provides you with an easy method for inspecting the assembly code generated for each C statement. For more information, refer to Section 2.5 on page 2-36.

- **Library-Build Utility**

  The compiler package has a utility that creates a runtime-support library and installs standard header files for any configuration of compiler options that you choose. For more information, see Chapter 6.

- **Flexible Assembly Language Interface**

  The compiler has straightforward calling conventions, enabling you to easily write assembly and C functions that call each other. For more information, refer to Chapter 4.

- **ROM-able Code**

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

- **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, refer to Section 2.2 on page 2-21.

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.

The C compiler includes an optimizer that allows you to produce highly optimized code. The optimizer is explained in Section 2.3.

The compiler package also includes a utility that interlists your original C source statements into the compiler's assembly language output. This enables you to inspect the assembly language code generated for each C statement. The interlist utility is explained in Section 2.5.

This chapter includes the following topics:

Topic **Page**

2.1 Compiling C Code

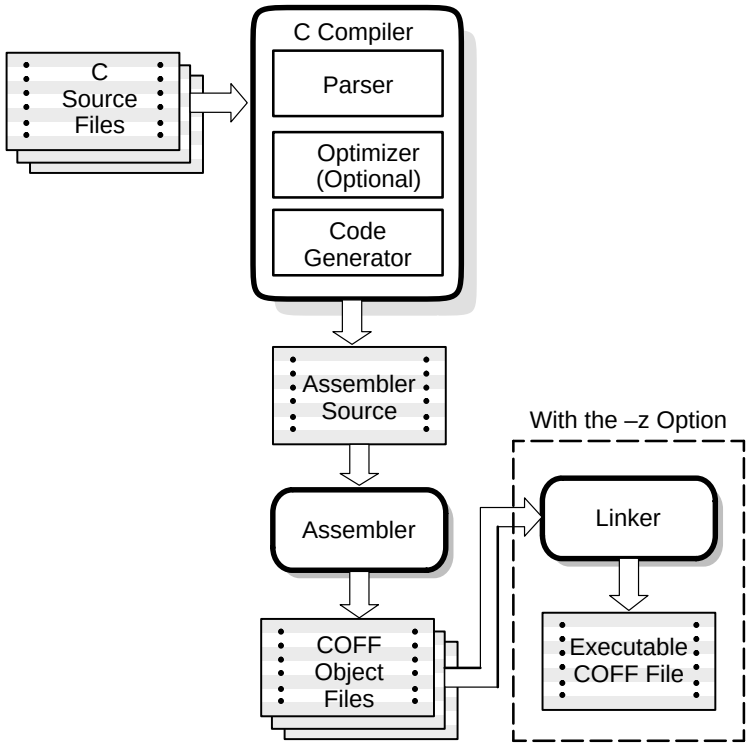
The dspcl shell program lets you compile, assemble, and optionally link in one step. The dspcl shell runs one or more source modules through the following:

- **Compiler** — The compiler includes the parser, the optimizer, and the code generator.
- **Assembler** — The assembler generates a COFF object file.
- **Linker** (optional) — Although you must link your files to create an executable object file, the linker is optional at this point. You can compile various files with the shell and link later.

For more information about the fixed-point assembler and linker, refer to the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

By default, dspcl compiles and assembles files; however, if you use the –z option, dspcl will compile, assemble, and link your files. Figure 2–1 illustrates the path dspcl takes with and without the –z option.

Figure 2–1. The Shell Program Overview



2.1.1 Invoking the C Compiler

To run the compiler, enter:

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

dspcl	is the command that invokes the compiler and assembler.
–options	affect the way the compiler, optimizer, or assembler 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.
link_options	affect the way the linker processes input files.
object files	names the object files that the compiler creates.

Options control the way the compiler processes files, and the filenames provide a method of identifying source files, intermediate files, and output files. The –z option and its associated information must follow all filenames and compiler options on the command line. For example, if you wanted to compile two files named symtab and file, assemble a third file named seek.asm, and use the quiet option (–q), you would enter

```
dspcl –q symtab file seek.asm
```

As dspcl encounters each source file, it prints the filename in square brackets [for C files] and angle brackets <for asm files>. The example above uses the –q option to suppress the additional progress information that dspcl produces. Entering the command above produces:

```
[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) 1987–1994, Texas Instruments Incorporated
   ″symtab.c″:  ==> main
   ″symtab.c″:  ==> lookup
TMS320C2x/2xx/5x ANSI C Codegen         Version X.XX
Copyright (c) 1987–1994, Texas Instruments Incorporated
   ″symtab.c″:  ==> main
   ″symtab.c″:  ==> lookup
DSP Fixed Point COFF Assembler          Version X.XX
Copyright (c) 1987–1994, Texas Instruments Incorporated
   PASS 1
   PASS 2

No Errors, No Warnings
```

2.1.2 Specifying Filenames

The input files specified on the command line can be C source files, assembly source files, or object files. dspcl uses filename extensions to determine the file type.

Extension	File Type
.c or none (.c assumed)	C source
.asm, .abs, or .s* (extension begins with s)	assembly source
.o* (extension begins with o)	object

Files without extensions are assumed to be C source files, and a .c extension is assumed.

You can use the –e option to change these default extensions, causing dspcl to associate different extensions with assembly source files or object files. You can also use the –f option on the command line to override these file type interpretations for individual files. For more information about the –e and –f options, refer to page 2-11.

The conventions for filename extensions allow you to compile C files and assemble assembly files with a single command, as shown in the example on page 2-3.

You can use wildcard characters to compile or assemble multiple files. Wildcard specifications vary by system; use the appropriate form. For example, to compile all the files in a directory, enter the following (DOS or OS/2 system):

```
dspcl *.*
```

2.1.3 Compiler Options

Command-line options control the operation of both dspcl and the programs it calls. This section provides a description of option conventions, an option summary table, and a detailed description of each of the options.

Options are either single letters or two-letter pairs, *are not* case sensitive, and 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, *–mr* and *–mb* can be combined as *–mrb*. Options that have parameters, such as *–uname* and *–i<dir>*, must be specified separately.

You can set up default options for dspcl by using the C_OPTION environment variable. For a detailed description of the C_OPTION environment variable, refer to subsection 2.1.4, page 2-19.

Table 2–1 summarizes all shell and linker options. The table is followed by in-depth descriptions of each of the options.

Table 2–1. Option Summary Table

General Shell Options	Option	Effect
These options control the overall operation of the dspcl shell program. For more information, see page 2-10.	–c	disable linking (negates –z)
	–d*name*[=def]	predefine a constant (or *name*)
	–g	enable symbolic debugging
	–i*dir*	define #include search path
	–k	keep .asm file
	–n	create .asm file but don't run assembler
	–q	suppress program messages (quiet)
	–qq	suppress all messages (super quiet)
	–r*register*	reserve global register
	–s	interlist C source with .asm source
	–u*name*	undefine *name*
	–v*xx*	determine processor: *xx* = –25, –50 or –2xx
	–z	enable linking
File Specifiers	**Option**	**Effect**
These options modify dspcl's default interpretation of filename extensions in determining how to process a file. For more information, see page 2-11.	–ea*ext*	set default assembly file extensions
	–eo*ext*	set default object file extensions
	–fa *file*	identify assembly language file (default for .asm or .s*)
	–fc *file*	identify C source file (default for .c or no ext)
	–fo *file*	specify object file (default for .o*)
	–fr*dir*	specify object file directory
	–ft	override TMP environment variable

Table 2–1. Option Summary Table (Continued)

Parser Options	Option	Effect
These options control the preprocessing, syntax-checking, and error-handling behavior of the compiler. For more information, see page 2-12.	–p?	enable trigraph expansion
	–pe	treat code-E errors as warnings
	–pf	generate function prototype listing file
	–pk	allow K&R compatibility
	–pl	generate preprocessed listing (.pp file)
	–pn	suppress #line directives in .pp file
	–po	preprocess only
	–pr	generate error listing
	–pw	suppress warning messages
Inlining Options	**Option**	**Effect**
These options control inlining of functions. For more information, see page 2-13.	–x0	disable inlining
	–x1	default inlining level
	–x2 (or x)	define _INLINE + invoke optimizer at level 2, if not already invoked
Type-Checking Options	**Option**	**Effect**
These options relax type-checking rules. For more information, see page 2-13.	–tf	relax prototype checking
	–tp	relax pointer combination checking
Runtime Model Options	**Option**	**Effect**
These options customize the executable output of the compiler for your specific application. For more information, see page 2-14.	–ma	assume aliased variables
	–mb	disable RPTK instruction
	–mn	enable optimizer options disabled by –g
	–mp	generate prolog/epilog inline
	–mr	list register-use information
	–ms	optimizer for code space
	–mx	avoid ’C5x silicon bugs

Table 2–1. Option Summary Table (Continued)

Assembler Options	Option	Effect
These options control the behavior of the assembler. For more information, see page 2-15 and Chapter 3 of the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*..	–aa	allow absolute listing directives
	–al	produce assembly language listing file
	–ap	enable 'C2x to 'C2xx or 'C5x port switch
	–app	enable 'C2x to 'C2xx port switch and define .TMS32025 and .TMS3202xx
	–as	keep labels as symbols
	–ax	produce cross-reference file
Linker Options	**Option**	**Effect**
These options control the behavior of the linker. For more information, see page 2-15 and Chapter 8 of the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.	–a	generate absolute output (default)
	–ar	generate relocatable output module
	–b	disable symbolic debugging data
	–c	use ROM initialization (default)
	–cr	use RAM initialization
	–e *global_sym*	define entry point
	–f *fill_value*	define fill value
	–h	make all global symbols static
	–heap *size*	set heap size (words)
	–i *dir*	define library search path
	–l *filename*	define library name
	–m *filename*	name the map file
	–o *filename*	name the output file
	–r	keep relocation entries in output module
	–s	strip symbol table
	–stack *size*	set stack size (words)
	–u *symbol*	undefine entry point
	–v0	generate version 0 COFF format
	–w	generate output section warning
	–x	force rereading of libraries

Table 2–1 . Option Summary Table (Continued)

Optimizer Options	Option	Effect
These options control the behavior of the optimizer. For more information, see page 2-17.	–o0	level 0 register optimization
	–o1	level 1 + local optimization
	–o2 (or –o)	level 2 + global optimization
	–o3	level 3 + file optimization
	–oe	assume no function in the module is called from an interrupt routine
	–oi*size*	set automatic inlining size (–o3 only)
	–ol0 (–oL0)	file alters a standard library function
	–ol1 (–oL1)	file defines a standard library function
	–ol2 (–oL2)	file does not define or alter library functions
	–on0	disable optimizer information file
	–on1	produce optimizer information file
	–on2	produce a verbose information file
	–op0	functions in other files may call functions and modify variables defined here
	–op1	functions in other files do not call functions defined here but may modify variables defined here (default)
	–op2	functions in other files do not call functions or modify variables defined here

General Shell Options

–c	suppresses the linking option; it causes dspcl not to run the linker even if –z is specified. This option is especially useful when you have –z specified in the C_OPTION environment variable and you don't want to link. For more information, refer to subsection 2.8.1, page 2-48.
–d*name*[**=***def*]	predefines the constant *name* for the preprocessor; equivalent to inserting **#define** *name def* at the top of each C source file. If the optional [*def*] is omitted, **–d***name* sets *name* equal to 1.
–g	causes the compiler to generate symbolic directives for use with the C source level debugger.
–i*dir*	adds *dir* to the list of directories the compiler searches for #include files. You can use this option a maximum of 10 times to define several directories; be sure to separate –i options with spaces. If you don't specify a directory name, the preprocessor ignores the –i option. For more information, refer to subsection 2.2.2, page 2-22.
–k	keeps the assembly language file. Normally, dspcl deletes the output assembly language file after compilation is finished, but using –k allows you to retain the assembly language output from the compiler.
–n	causes dspcl to compile only. If you use –n, the specified source files are compiled but not assembled or linked. This option overrides –z and –c options.
–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 interlists C source statements into the assembly language output of the compiler, allowing you to inspect the code generated for each C statement. This option automatically uses the –k option. For more information, refer to Section 2.5, page 2-36.
–u*name*	undefines the predefined constant *name*. 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 enables the linking option; it causes dspcl to run the linker on specified object files. The –z option must follow all source files and compiler options on the command line. All arguments that follow –z on the command line are passed to, and interpreted by, the linker. For more information, refer to subsection 2.8.2, page 2-49.

File Specifiers

–e overrides dspcl's default naming conventions for filename extensions on assembly files and object files. The two –e options are listed below.

–ea [.]*asmext* for assembly files (default is .asm)

–eo [.]*objext* for object files (default is .obj)

For example:

```
dspc1 –ea .rrr –eo .o37 fit.rrr
```

assembles the file fit.rrr and creates an object file named fit.o37.

The "." in the extensions and the space between the option and the extension are optional (the example could have been –earrr –eoo37…).

The –e option affects both the interpretation of filenames on the command line and the names of the output files and should always precede any filename on the command line.

–f overrides default interpretations for source file extensions. If your naming conventions do not conform to those of dspcl, you can use –f options to specify which files are C source files, assembly language files, and object files. You can insert an optional space between the –f option and the filename. The –f options are:

–fa*file* for assembly language source file

–fc*file* for C source file

–fo*file* for object file

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

```
dspc1 –fc cfile.s –fa assy
```

Note that –f cannot be applied to a wildcard file specification.

–fr permits you to specify a directory for object files. If the –fr option is not specified, the shell places object files in the current directory. To specify an object file directory, insert the directory's pathname on the command line after the –fr option:

```
dspcl –fr d:\object ...
```

–ft permits you to specify a directory for temporary intermediate files. The –ft option overrides the TMP environment variable (described in subsection 2.1.5, page 2-20). To specify a temporary directory, insert the directory's pathname on the command line after the –ft option:

```
dspcl –ft d:\temp ...
```

Parser Options

–p? enables trigraph expansion. Trigraphs are special escape sequences of the form

```
??c
```

where c is a character. The ANSI C standard defines these sequences for the purpose of compiling programs on systems with limited character sets. By default, the compiler does not recognize trigraphs; use –p? to enable trigraphs. For more information, refer to the ANSI C specification, § 2.2.1.1, or K&R A12.1.

–pe causes the parser to treat code-E errors as warnings, allowing complete compilation. Normally, the code generator does not run if the parser detects any code-E errors. Note that the code-F errors are always fatal. For more information, see Section 2.6, page 2-38.

–pf produces a function prototype listing file, which contains the prototype of every procedure in all corresponding C files. Each function prototype file is named like its corresponding C file with a .pro extension. The –pf option is useful when you are conforming code to the ANSI C standard or generating a listing of procedures defined.

–pk relaxes certain requirements newly imposed by the ANSI C standard (that are stricter than those required by earlier K&R compilers). This facilitates compatibility between K&R C programs and the TMS320C2x/C2xx/C5x ANSI compiler. The effects of the –pk option are described in Section 3.7, page 3-12.

–pl generates a preprocessed listing file. The compiler writes a modified version of the source file to an output file called file.pp. The .pp file contains all the source from #include files and expanded macros; it does not contain any comments or code for false #if or #ifdef directives. The only remaining preprocessor directive is #line. For more information, refer to subsection 2.2.3, page 2-24.

–pn suppresses line and file information. The –pn option causes the #line directives of the form

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

to be suppressed in the .pp file generated with –po or –pl. The –pn option is sometimes useful when you are compiling machine-generated source.

–po runs the compiler for preprocessing only. When invoked with the –po option, the compiler processes only macro expansions, included files, and conditional compilation. The compiler writes the preprocessed file with a .pp extension. For more information, refer to subsection 2.2.3, page 2-24.

–pr creates a parser error message file. The error file has the base name of the input file and an .err extension. The file contains all error messages generated by the parser.

–pw suppresses warning messages (code-W errors). Rather than producing warning messages, the compiler produces diagnostic messages for only those errors that prevent complete compilation. This option can be doubled (–pww) to suppress recoverable messages (code-E errors) also. For more information, refer to Section 2.6, page 2-38.

Inlining Options

–x*n* controls inline function expansion done by the optimizer when functions have been defined or declared as *inline*. The possibilities are **–x0,** which disables all inlining; **–x1**, which inlines all intrinsic operators; and **–x2** (or just –x). Note that –x1 is the default inlining option; it occurs whether or not the optimizer is invoked. The last option may be specified as –x or –x2 interchangeably. It invokes the optimizer at level 2, if inlining has not already been invoked, and defines the _INLINE preprocessor symbol. See Section 2.4, page 2-30 for more information.

Type-Checking Options

–tf relaxes type checking on redeclarations of prototyped functions. In ANSI C, if a function is declared with an old-format declaration, such as

```
int func();
```

and then later declared with a prototype, such as

```
int func(float a, char b);
```

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). With the –tf option, the compiler overlooks such redeclarations of parameter lists.

–tp relaxes 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 pointers are passed to prototyped functions and when the passed pointer type would ordinarily disagree with the declared parameter type in the prototype.

Runtime-Model Options

–ma assumes variables are aliased. The compiler assumes that pointers may alias (point to) named variables and therefore 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 maybe a preferable tradeoff if your system uses a cheaper form of memory for program memory space than it does for data memory space.

–mn re-enables 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 and that either implements interrupts or is compiled with optimization.

There is one problem that this switch does not work around. When you run the compiler 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 will not occur. See the latest silicon errata sheet for more information.

Assembler Options

For more information about assembler options, refer to the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User’s Guide.*

–aa invokes the assembler with the –a option to allow absolute listing directives in the input file. The assembler does not produce an object file when this option is used.

–al invokes the assembler with the –l (lowercase L) option to produce an assembly language 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 .TMS320C2xx assembler symbols. Use –app with the –v2xx option.

–as retains labels. Label definitions are written to the COFF symbol table for use with symbolic debugging.

–ax invokes the assembler with the –x option to include cross-reference information in the listing file.

Linker Options

All command-line input following –z is passed to the linker as parameters or options. For more information about linker options, refer to the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User’s Guide.*

–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	enables linking conventions defined by the ROM autoinitialization model of the TMS320C2x/2xx/5x C compiler.
–cr	enables linking conventions defined by the RAM autoinitialization model of the TMS320C2x/2xx/5x C compiler.
–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.
–h	makes all global symbols static.
–heap *size*	sets heap size (for the dynamic memory allocation in C) to *size* words and defines a global symbol that specifies the heap size. Default = 1K words.
–i *dir*	alters the library-search algorithm to look in *dir* before looking in the default location. This option must appear before the –l option. The directory must follow operating system conventions.
–l *filename*	names an archive library file as linker input; *filename* 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.
–o *filename*	names the executable output module. The default *filename* is a.out and must follow operating system conventions.
–q	suppresses 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 = 1K words.
–u *symbol*	places the unresolved external symbol *symbol* into the output module's symbol table.
–v0	generates version 0 COFF format.

–w generates a warning when an output section that is not specified with the SECTIONS directive is created.

–x forces rereading of libraries and resolves back references.

Optimizer Options

–o*n* causes the compiler to optimize the intermediate file that is produced by the parser. The *n* denotes the level of optimization. There are four levels of optimizations: **–o0**, **–o1**, **–o2,** and **–o3**.

If you do not indicate a level (0, 1, 2, 3) after the –o option, the optimizer defaults to level 2. For more information about the optimizer, refer to Section 2.3, page 2-25 and Appendix A.

–oe 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. This option automatically invokes the optimizer at level two.

–oi*size* controls automatic inlining of functions (not defined or declared as *inline*) at optimization level 3. You specify the *size* limit for the largest function that will be inlined. If the –oi option is not used, the optimizer will inline very small functions when invoked at level 3. Setting the size to 0 (–oi0) disables automatic inlining completely. Note that the –x option controls inlining of functions declared with the *inline* keyword.

–ol*n* (lowercase L) controls file level optimizations. When you invoke the optimizer at level 3 (–o3), some of the optimizations use known properties of the standard library functions. If the file you are compiling redefines any of these standard functions, the compiler may produce incorrect code. Use the –ol option to notify the optimizer if any of the following situations exist:

- **–ol0** : This file defines a function with the same name as a standard library function.
- **–ol1** : This file contains the standard library definition functions for those functions that are defined in it.
- **–ol2** : The file does not alter standard library functions. Use this option to restore the default behavior of the optimizer if you have used one of the other two –ol options in a command file, an environment variable, etc.

–on*n* causes the compiler to produce a user-readable optimization information file with a .nfo extension. This option works only when the –o3 option is used. There are three levels available:

- **–on0**: Do not produce an information file. Use this option to restore the default behavior of the optimizer if you have used one of the other two –on options in a command file, an environment variable, etc.
- **–on1**: Produce an optimization information file.
- **–on2**: Produce a verbose optimization information file.

–op*n* specifies whether functions in other files can call this file's EXTERN functions or modify this file's EXTERN variables. Level 3 optimization combines this information with its own file-level analysis to decide whether to treat this file's EXTERN function and variable definitions as if they had been declared STATIC. The following three levels are defined:

- **–op0**: Signals the optimizer that functions in other modules might call functions or variables defined in the current module. This disables some of the –o3 optimizations.
- **–op1**: Signals the optimizer that no functions exist in other modules that might call functions defined in this module, and that no interrupt function defined elsewhere might call functions defined here. This is the default when –o3 is used.
- **–op2**: Signals the optimizer that no functions in this module are called by other modules and no variable declared in this module will be altered by another module.

Level –op1 reverts to –op0 if the file does not define the function *main* or an interrupt function or if it contains calls to unknown functions. This level is the default because it is unlikely that an interrupt function defined elsewhere would call a user function defined in a file that contains main and all of the user functions called directly or indirectly from main.

Level –op2 also reverts to –op0 if no main or interrupt functions are found, but unlike level 1, it does not revert if there are calls to unknown functions.

Use of the –op options automatically invokes the optimizer at level 3 (–o3).

2.1.4 Using the C_OPTION Environment Variable

An environment variable is a system symbol that you define and assign to a string. You may find it useful to set dspcl default options using the C_OPTION environment variable; if you do, these default options and/or input filenames are used every time you run dspcl.

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

Options in the environment variable 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, enable symbolic debugging, and link, then set up the C_OPTION environment variable as follows.

Host	Enter
DOS or OS/2	`set C_OPTION= – qg –z`
UNIX	`setenv C_OPTION ″–qg –z″`

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

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

2.1.5 Using the TMP Environment Variable

The dspcl creates intermediate files as it processes your program. For example, the parser phase of dspcl creates a temporary file used as input by the code generation phase. By default, dspcl puts intermediate files in the current directory. However, you can name a specific directory for temporary files.

This feature allows use of a RAM disk or other high-speed storage files. 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. There are two ways to specify a temporary directory:

- Use the TMP environment variable:

  ```
  set TMP=d:\temp
  ```

  This example is for a PC. Use the appropriate command for your host.

- Use the –ft option on the command line:

  ```
  dspcl –ft d:\temp....
  ```

  The –ft option, if used, overrides the TMP environment variable.

2.2 Preprocessing C Code

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)

This section describes specific features of the TMS320C2x/C2xx/C5x preprocessor. A general description of C preprocessing is in Section A12 of K&R.

2.2.1 Predefined 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 optimizer option; undefined otherwise

† Specified by the ANSI standard

You can use these names in the same manner as any other defined name. For example,

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

could translate to a line such as:

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

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

2.2.2 #include File Search Paths

The #include preprocessor directive tells the compiler to read source statements from another file. The syntax for this directive is:

#include "*filename*" **or** **#include** <*filename*>

The *filename* names the #include file that the compiler reads statements from; you can enclose the filename in double quotes or in angle brackets. The filename can be a complete pathname, have partial path information, or have no path information.

If you enclose the filename in *double quotes*, the compiler searches for the file in the following directories, in the order given:

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 option in dspcl
3) Directories set with the environment variable C_DIR

If you enclose the filename in *angle brackets*, the compiler searches for the file in the following directories, in the order given:

1) Directories named with the –i option in dspcl
2) Directories set with the environment variable C_DIR

Note: Enclosing the Filename in <angle brackets>

If you enclose the filename in angle brackets, the compiler *does not* search for the file in the *current directory*.

Include files are sometimes stored in directories. You can augment the compiler's directory search algorithm by using the –i shell option or the environment variable C_DIR to identify a directory name.

–i dspcl Option

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

dspcl –i *pathname ...*

You can use up to 10 –i options per invocation; each –i option names one *pathname*. In C source, you can use the #include directive without specifying any path information for the file; instead, you can specify the path information with the –i option. For example, assume that a file called source.c is in the current directory. The file, source.c, contains :

```
#include "alt.h"   or   #include <alt.h>
```

The table below lists the complete pathname for alt.h and shows how to invoke the compiler. Select the row for your host system.

Host	Pathname for alt.h	Invocation Command
DOS or OS/2	`c:\dsp\files\alt.h`	`dspcl –ic:\dsp\files source.c`
UNIX	`/dsp/files/alt.h`	`dspcl –i/dsp/files source.c`

The included filename is enclosed in double quotes. The compiler first searches for alt.h in the current directory, because source.c is in the current directory. Then, the compiler searches the directory named with the –i option.

C_DIR Environment Variable

The compiler uses the environment variable **C_DIR** to name alternate directories that contain #include files. To specify the same directory for #include files, as in the previous example, set C_DIR with one of these commands:

Host	Enter
DOS or OS/2	`set C_DIR=c:\dsp\files`
UNIX	`setenv C_DIR "/dsp/files"`

Then you can include alt.h with one of the following directive statements:

`#include "alt.h"` or `#include <alt.h>`

Then invoke the compiler without the –i option:

```
dspcl source.c
```

This causes the compiler to use the path in the C_DIR environment variable to find the #include file.

The pathnames specified with C_DIR are directories that contain #include files. You can separate pathnames with a semicolon or with blanks.

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

Host	Enter
DOS or OS/2	`set C_DIR=`
UNIX	`unsetenv C_DIR`

2.2.3 Generating a Preprocessed Listing File (–pl , –pn, –po Options)

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

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

(These functions correspond to translation phases 1–3 as 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. If you use the –pn option, no #line directives are inserted.

If you use the –po option, the compiler performs *only* the preprocessing functions listed above and then writes out the preprocessed listing file; no syntax checking or code generation takes place. The –po option can be useful when you debug macro definitions or when host memory limitations dictate separate preprocessing (refer to *Parsing in Two Passes* on page 2-43). The resulting preprocessed listing file is a valid C source file that can be rerun through the compiler.

2.2.4 #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, which, like #error, forces a diagnostic message but does not halt compilation. The syntax of #warn is identical to that of #error. For more information, refer to Section A12.7 of K&R.

2.3 Using the Optimizer

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

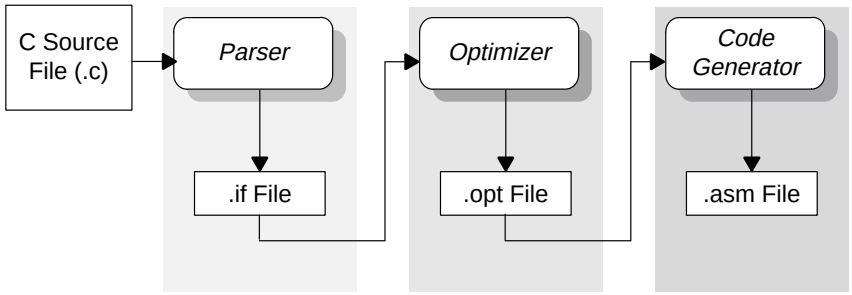
The optimizer runs as a separate pass between the parser and the code generator. The easiest way to invoke the optimizer is to use the dspcl shell program, specifying the –o option on the dspcl command line. You can specify the optimization level (0, 1, 2, or 3) after the –o option if you wish. The default level is 2.

For example, to invoke the compiler using level 2 optimization, enter:

```
dspcl –o function.c
```

Figure 2–2 illustrates the execution flow of the compiler with standalone optimization.

Figure 2–2. Compiling a C Program With the Optimizer



The optimizer also recognizes dspcl options –s, –ma, –q, and –pk; these options are discussed in subsection 2.1.3, page 2-5.

For information on how to invoke the optimizer outside dspcl, refer to subsection 2.7.2, page 2-44.

2.3.1 Optimization Levels

There are four levels of optimization: 0, 1, 2, and 3. These levels control the type and degree of optimization.

- **Level 0**
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates dead code
  - Simplifies expressions and statements
  - Expands calls to functions declared *inline*

❑ **Level 1**

Performs all level 0 features, plus:

- Performs local copy/constant propagation
- Removes dead assignments
- Eliminates local common subexpressions

❑ **Level 2**

Performs all level 1 features, plus:

- Performs loop structure optimizations
- Eliminates global common subexpressions
- Eliminates global dead assignments
- Performs loop unrolling

❑ **Level 3**

Performs all level 2 features, plus:

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

Note: Files That Redefine Standard Library Functions

The optimizer uses known properties of the standard library functions to perform level 3 optimizations. If you have files that redefine standard library functions, use the –ol (lowercase L) options to inform the optimizer. (See page 2-17.)

The above list describes optimizations performed by the standalone optimization pass. The code generator performs several additional optimizations, particularly TMS320C2x/C2xx/C5x-specific optimizations; it does so regardless of whether you invoke the optimizer. These optimizations are always enabled and are not affected by the optimization level you choose.

For more information about the meaning and effect of specific optimizations, refer to Appendix A.

2.3.2 Definition-Controlled Inline Expansion Option (–x Option)

When the optimizer is invoked, the –x optimizer option controls inline expansion of functions that have been declared as *inline* by inhibiting or allowing the expansion of their code in place of calls. That is, code for the function will be inserted (inlined) into your function at each place it is called whenever the optimizer is invoked and the –x option is not equal to –x0. The –x2 option automatically invokes the optimizer at the default level (level 2, if the –o option is not specified separately) and defines the _INLINE preprocessor symbol as equal to 1, which causes expansion of functions declared as inline and controlled by the _INLINE symbol (For more information about _INLINE, see subsection 2.4.3, page 2-33).

Inlining makes a program faster by eliminating the overhead caused by function calls, but inlining sometimes increases code size.

For more information, see Section 2.4, page 2-30.

2.3.3 Using the Optimizer With the Interlist Option

Optimization makes normal source interlisting impractical because the optimizer extensively rearranges your program. Therefore, the optimizer writes reconstructed C statements (as assembly language comments), which show the optimized C statements. The comments also include a list of the allocated register variables. Note that occasionally the optimizer interlist comments may be misleading because of copy propagation or assignment of multiple or equivalent variables to the same register.

2.3.4 Debugging Optimized Code

Ideally you should debug a program in an unoptimized form and reverify its correctness after it has been optimized. The debugger can be used with optimized code, but the 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 on dspcl; –mn re-enables the optimizations disabled by –g.

2.3.5 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 to be used with the optimizer, you should note the following special considerations to ensure that your program performs as you intend.

asm Statements

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

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. The compiler won't 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 will be optimized down to a single memory read. To correct this, declare `ctrl` as:

```
volatile unsigned int *ctrl
```

Aliasing

Aliasing occurs when a single object can be accessed 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 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, but that will not make its address available 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's address. In cases where this assumption is invalid, use the –ma option in dspcl to force the compiler to assume worst-case aliasing. In worst-case aliasing, any indirect reference may refer to such a variable.

2.4 Function Inlining

When an inline function is called, the code for the function is inserted at the point of the call. This is advantageous in short functions for two 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 expansion is performed in one of three ways.

- The intrinsic operators of the target system (such as abs) are inlined by the compiler by default. This happens whether or not the optimizer is used and whether or not any compiler or optimizer options are used. (You can defeat this automatic inlining by invoking the compiler with the –x0 option.)
- Definition-controlled inline expansion is performed when two conditions exist:
  - The *inline* keyword is encountered in source code.
  - The optimizer is invoked (at any level).

  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 defined as static inline, expansion occurs despite the presence of local statics. 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 only a few times (though the compiler does not enforce this).

  You can control this type of function inlining in two ways:

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

- By *defining* 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, but the function is inlined only within the module where it is defined as inline. It is called by other modules unless they contain a compatible static inline declaration. Functions defined as inline are expanded when the optimizer is invoked and the –x option is not equal to –x0. Setting the –x option to –x2 automatically invokes the optimizer at the default level (level2).

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

- By *declaring* a function as static inline, you can specify that the function is inlined in the present module. This names the function and specifies that the function is to be expanded 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. Declaring a function as static inline in a header file specifies that the function is inlined in any module that includes the header.

  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 runtime-library functions, are expanded whenever the optimizer is invoked and the _INLINE preprocessor symbol is equal to 1. When you define an inline function, 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 will be unresolved. For more information, see subsection 2.4.3, page 2-33.

- Automatic inline expansion (of functions *not* declared as inline) is done when the optimizer is invoked at level 3. By default, the optimizer will inline very small functions. You can change the size of functions that are automatically inlined with the **–oi***size* option. The –oi option specifies that functions whose size, times number of calls, is less than *size* units are inlined regardless of how they were declared. The optimizer measures the size of a function in arbitrary units. However, the size of each function is reported in the optimizer information file (–on1 option). If you want to be certain that a function is always inlined, use the inline keyword (discussed above and in subsection 2.4.2, page 2-32). You can defeat all automatic inlining of small functions not declared as inline by setting the size to 0 (–oi0).

Note: Function Inlining Can Greatly Increase Code Size

Expanding functions inline expands code size and inlining a function that is called a great number of times can expand code size exponentially. Function inlining is optimal for functions that are called only a small number of times, and for small functions that are called more often. If your code size seems too large, try compiling with the –x0 option and note the difference in code size.

2.4.1 Automatic Inline Expansion Option (–oi*size* Option)

The optimizer will automatically inline all small functions (not defined or declared with the *inline* keyword) when invoked at level 3. A command-line option controls the size of functions inlined when the optimizer is invoked at level 3. The –oi option can be used three ways:

- If you set the *size* parameter to 0 (–oi0), all size-controlled inlining is disabled.
- If you do not use the –oi option, the optimizer inlines very small functions.
- If you set the *size* parameter to a nonzero integer, the optimizer inlines all functions whose size is less than the *size* parameter. If the function is called more than once, the optimizer multiplies the size of the function by the number of calls and inlines the function only if the resulting product is less than the *size* parameter. The optimizer measures the size of a function in arbitrary units. The optimizer information file (created with the –on1 or –on2 option), however, reports the size of each function in the same units that the –oi option uses.

2.4.2 Controlling Inline Expansion (–x Option)

A command-line switch controls the types of inline expansion performed.

–x0 causes no inline expansion. This option defeats the default expansions listed below.

–x1 is the default value. The intrinsic operators are inlined wherever they are called. This is true whether or not the optimizer is invoked, and whether or not a –x option is specified (except –x0). Intrinsic operators are:
- abs
- labs
- fabs

–x2/–x creates the preprocessor symbol _INLINE, assigns it the value 1, and invokes the optimizer at level 2, thereby enabling definition-controlled inline expansion.

If a function has been defined or declared as inline, it is expanded inline whenever the optimizer is called and the –x option is not equal to –x0. Setting the –x option to –x2 automatically invokes the optimizer and thus causes the automatic expansion of functions defined or declared as inline, as well as causing the other optimizations defined at level 2, which is the default level for the optimizer.

The _INLINE preprocessor symbol has been used to control the expansion of the runtime-support library modules. They are expanded if _INLINE is equal to 1, but they are called if _INLINE is not equal to 1. Other functions can be set up to use _INLINE in the same way. For more information, see subsection 2.4.3.

If the –x, –x1 or –x2 option is used together with the –o option at any level, the optimizer is invoked at the level specified by –o rather than at the level specified by –x.

2.4.3 _INLINE Preprocessor Symbol

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

The _INLINE symbol is used in the string.h header file to declare the function correctly, regardless of whether inlining is used. The _INLINE symbol is turned off in the memcpy source *before* the header file is included, because it is unknown whether the rest of the module is compiled with inlining.

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

Example 2–1 on page 2-34 illustrates how the runtime-support library uses the _INLINE symbol. You want to use the _INLINE preprocessor symbol in the same way so that your programs run regardless of whether inlining is selected for any or all of the modules in your program.

Example 2–1. How the Runtime-Support Library Uses the _INLINE Symbol

```
/*****************************************************************************/
/* 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
```

```
/*****************************************************************************/
/* 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);
}
```

There are two definitions of the memcpy 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 (which is in memcpy.c) is for the library so that the callable version of memcpy exists when inlining is disabled. Since this is not an inline function, the _INLINE symbol is undefined (#undef) before string.h is included so that the noninline version of memcpy's prototype is generated.

If the application is compiled with the –x option *and* the string.h header is included, all references to memcpy in the runtime-support library are inlined and the linker does not have to use the memcpy in the runtime-support library to resolve any references. Any modules that call memcpy and are not compiled with inlining enabled will generate calls that the linker resolves by getting the memcpy code out of the library.

2.5 Using the Interlist Utility

The compiler package includes a utility that interlists your original C source statements into the assembly language output of the compiler. The interlist utility enables you to inspect the assembly language code generated for each C statement.

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

```
dspcl –s function
```

The interlist utility runs 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 language file as comments (each of which begins with *;>>>>)*. The output assembly language file, function.asm, is assembled normally. The –s option automatically prevents dspcl from deleting the interlisted assembly language file (as if you had used –k).

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 subsection 2.7.4, page 2-47.

Note: Using the –s Option With the Optimizer

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

2.6 How the Compiler Handles 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 uppercase letter identifies the error class; a 3-digit number uniquely identifies the error.
error message	is the text of the message.

Errors are categorized into four classes according to the severity of the error; these classes are identified by the letters *W*, *E*, *F*, and *I* (upper case i):

- **Code-W errors** are warnings: they result from a condition that is technically undefined according to the rules of the language and may not generate what you intended. This is an example of a code-W error:

```
"file.c", line 42: [W063] illegal type for register variable 'x'
```

- **Code-E errors** are recoverable: they result 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. Refer to subsection 2.6.1, page 2-39, for more information. This is an example of a code-E error:

```
"file.c", line 66: [E056] illegal storage class for function 'f'
```

- **Code-F errors** are fatal: they result from a condition that violates the syntactic or semantic rules of the language. The compiler cannot recover and therefore does not generate output for code-F errors. This is an example of a code-F error:

```
"file.c", line 71: [F090] structure member 'a' undefined
```

- **Code-I errors** are implementation errors: they occur 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 Table 3–2, page 3-15.) This is an example of a code-I error:

```
"file.c", line 99: [I015] block nesting too deep (max=20)
```

Other user errors, such as incorrect command-line syntax or inability to find specified files, are also reported by the compiler. These errors are usually fatal and are identified by the symbol >> preceding the message. This is an example of such an error:

```
>> Cannot open source file 'mystery.c'
```

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

A *fatal error* is an error that prevents the compiler from generating an output file. Normally, E, F, and I errors are fatal, while W errors are not. The –pe option (used on the shell command line) causes the compiler to effectively treat E errors as warnings and, hence, to generate 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 may not generate what you expect.

There is no way to specify recovery from code-F or -I errors; these are always fatal and prevent generation of a compiled output file.

2.6.2 Suppressing Warning Messages (–pw Option)

The –pw option enables you to suppress the output of warning messages, causing the compiler to quietly ignore code-W errors (warnings). This is useful when you are aware of the condition causing the warning and consider it innocuous.

You can combine –pw options to suppress the more severe errors as well. Successive –pw options suppress errors of type W and E, respectively. Doubling –pw is useful in conjunction with the –pe option so that code-E errors are ignored completely. For example:

```
dspcl –peww *.c  ; completely ignore all W and E errors
```

2.6.3 An Example of How You Can Use Error Options

The following example demonstrates how the –pe and –pw options can be used to suppress errors and error messages. The examples use this 3-line code segment:

```
int *pi;
char *pc;
pi = pc;
```

- If you invoke the code with dspcl (and –q), 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 code with dspcl and 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 code with dspcl and –peww (–pe and double –pw), this is the result:

```
[err]
```

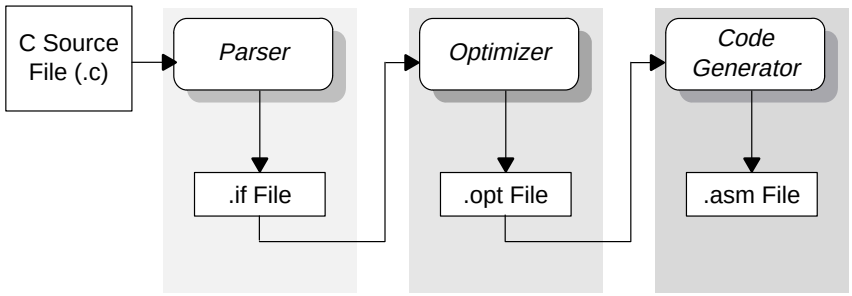
  As in the previous case, –pe causes the compiler to overlook the error and generate code. Because the two –pw options are used, the message is suppressed.

2.7 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 of the tools 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 made up of three distinct programs: the parser, the optimizer, and the code generator.

Figure 2–3. Compiler Overview



The input for the **parser** is a C source file. The parser reads the source file, checking for syntax and semantic errors, and writes out an internal representation of the program called an intermediate file. Subsection 2.7.1 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. Section 2.3, page 2-25, 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. Subsection 2.7.3, page 2-45, 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 2.5 on page 2-36 and subsection 2.7.4 on page 2-47 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. Section 2.8, page 2-48, describes how to run the linker. The linker is described fully in the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide.*

2.7.1 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 don't supply an extension, the parser assumes that the file's extension is *.c*. If you don't specify an input file, the parser prompts you for one.

output file names the intermediate file that the parser creates. If you don't 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 2–3 shows the parser options, the dspcl shell options, and the corresponding functions.

Table 2–3. 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 standalone 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.

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 may be able to work around host memory limitations by running the parser as two separate passes—the first pass preprocesses the file, and the second pass parses the file.

When you run the parser as one pass, it uses host memory to store both macro definitions and symbol definitions simultaneously. But when you run the parser as two passes, these functions can be separated. The first pass performs only preprocessing; therefore, memory is needed only for macro definitions. In the second pass, there are no macro definitions; therefore, 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, refer to Section 2.2, page 2-21.

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.

2.7.2 Invoking the Optimizer

The second step in compiling a TMS320C2x/C2xx/C5x C program — optimizing — is optional. 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 don't specify an input file, the optimizer prompts you for one.

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

options affect the way the optimizer processes the input file. The options that you use in standalone optimization are the same as those used for dspcl. Table 2–4 shows the optimizer options, the dspcl shell options, and the corresponding functions. Section 2.3, page 2-25 provides a detailed list of the optimizations performed at each level.

Table 2–4. 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*n*	–oL*n*	control assumptions about library function calls
–i*nn*	–oI*nn*	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
–o2	–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 3.5, page 3-8, for more information.

2.7.3 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 standalone 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 don't supply an extension, the code generator assumes that the extension is *.if*. If you don't 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 don't 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 don't 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 standalone code generator mode has a corresponding dspcl shell option that performs the same function. The following table shows the code generator options, the dspcl shell options, and the corresponding functions.

Table 2–5. 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 3.5, page 3-8, 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.

2.7.4 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 don't 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.

2.8 Linking C Code

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

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

This section describes how to invoke the linker with each method. It also discusses special requirements of linking C code, including selecting a runtime-support library, specifying the initialization model, and allocating the program into memory.

2.8.1 Invoking the Linker

The examples in this subsection show how to invoke the linker in a separate step after you have compiled or assembled your programs.

This is the general syntax for linking C programs in a separate step:

dsplnk –c *filenames* **–o** *name.out* **–l** *libraryname*
or
dsplnk –cr *filenames* **–o** *name.out* **–l** *libraryname*

dsplnk is the command that invokes the linker.

–c/–cr are options that tell the linker to use special conventions defined by the C environment. Note that when you use dspcl to link, dspcl passes –c to the linker by default.

filenames are object files created by compiling C programs or assembling assembly language programs.

–o *name.out* names the output file. If you don't use the –o option, the linker creates an output file with the default name of *a.out*.

–l *libraryname* identifies the appropriate archive library containing C runtime-support and floating-point math functions. (The –l option tells the linker that a file is an object library.) If you're linking C code, you must use a runtime-support library. The source for these runtime-support libraries is included with the C compiler; the library you use depends on which processor you use. You must use the library-build utility described in Chapter 6 to create a library for your application. Table 2–6 lists the fixed-point source libraries.

When you specify a library as linker input, the linker includes and links only those library members that resolve undefined references.

Table 2–6. Runtime-Support Source Libraries

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

For more information about the source libraries and their contents, refer to Section 5.1, page 5-2.

The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS linker directives to customize the allocation process. These directives are described in the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*. You can also use other linker options, listed in Table 2–1 on page 2-8.

2.8.2 Using dspcl to Invoke the Linker (–z Option)

The options and parameters discussed in this section apply to both methods of linking; however,when you are linking with dspcl, the options follow the –z shell option.

By default, dspcl does not run the linker. However, if you use the **–z** option, dspcl compiles, assembles, and links in one step. When using –z to enable linking, remember that:

- –z must follow all source files and compiler options on the command line (or be specified with C_OPTION)
- –z divides the command line into compiler options (before –z) and linker options (following –z)
- –c suppresses –z, so do not use –c if you want to link

All arguments that follow –z on the command line are passed 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 lnk.cmd –o prog.out –l rts25.lib
```

First, dspcl compiles all the files with .c extensions by using the –s and –q options. Second, because –z is specified, the linker links the resulting object files by using the linker command file *lnk.cmd*; the –o option names the output file.

The order in which the linker processes arguments can be important, especially for command files and libraries. When you use dspcl to run the linker, it passes arguments to the linker in the following order:

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

-c Shell Option

You can override the –z option by using the –c shell option. This option is helpful when you have specified –z in the C_OPTION environment variable and want to selectively disable linking with –c on the command line. Note that the –n option overrides both the –c and –z options.

The –c linker option has a different function from, and is independent of, the –c dspcl option. By default, dspcl automatically uses the –c linker option that tells the linker to use C source linking conventions (ROM model of initialization). If you want to use the RAM model of initialization, use the –cr linker option.

2.8.3 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 runtime-support library.
- ❏ Specify the initialization model.
- ❏ Determine how you want to allocate your program into memory.

This subsection 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, refer to the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide.*

Runtime-Support Libraries

All C programs must be linked with a runtime-support library. This archive library contains standard C library functions (such as malloc and strcpy) as well as functions used by the compiler to manage the C environment. To link in a library, simply use the –l option on the command line:

```
dsplnk –c  filenames    –l  rts25.lib ... or
dsplnk –cr  filenames   –l  rts25.lib ...
```

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

Generally, the libraries should be specified last 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 will not be resolved. You can use the –x option to force the linker to reread all libraries until references are resolved. Wherever you specify a library as linker input, the linker includes and links only those library members that resolve undefined references.

All C programs must be linked with an object module called *boot.obj*. When a program begins running, it executes boot.obj first. boot.obj contains code and data for initializing the runtime environment; the linker automatically extracts boot.obj and links it when you use –c or –cr and include rts<xx>.lib in the link.

Chapter 5 describes additional runtime-support functions that are included in rts<xx>.lib. These functions include ANSI C standard runtime support.

Initialization (RAM and ROM Models)

The C compiler produces tables of data for autoinitializing global variables. Subsection 4.8.2, page 4-29, discusses the format of these tables. These tables are in a named section called *.cinit*. The initialization tables can be used in either of two ways:

- **RAM model** (–cr linker option)

  Global variables are initialized at *load time*; use the –cr linker option. For more information about the RAM model, refer to *Initializing Variables in the RAM Model* on page 4-30.

- **ROM model** (–c linker option)

  Global variables are initialized at *runtime*; use the –c linker option. For more information about the ROM model, refer to *Initializing Variables in the ROM Model* on page 4-31.

The –c and –cr Linker Options

When you link a C program, you must use either the –c or the –cr option. These options tell the linker to use special conventions required by the C environment; for example, they tell the linker to use the ROM or RAM model of autoinitialization. When you use dspcl to link programs, the –c option is the default. 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; this ensures that boot.obj is automatically linked in from the runtime-support library rts<xx>.lib.

- The **.cinit** output section is padded with a termination record so that the loader (RAM model) or the boot routine (ROM model) knows when to stop reading the initialization tables.

- In the RAM model (–cr option),

  - The linker sets the symbol cinit to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at runtime.

  - 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.

    Note that a loader is not included as part of the C compiler package.

- In the ROM model (–c option), the linker defines the symbol cinit as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.

- The .const section is placed on the data page (page 1).

Sections Created by the Compiler

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

There are two basic kinds of sections created by the compiler: initialized and uninitialized. Table 2–7 summarizes the sections.

Table 2–7. Sections Created by the Compiler

Name	Type	Contents
.text	Initialized	Executable code and floating-point constant
.cinit	Initialized	Tables for explicitly initialized global and static variables
.const	Initialized	String literals, and global and static const variables that are explicitly initialized
.switch	Initialized	Switch statement tables
.bss	Uninitialized	Global and static variables
.stack	Uninitialized	Software stack area
.sysmem	Uninitialized	Dynamic memory area for malloc functions

When you link your program, you must specify where to locate the sections in memory and the amount of memory to allocate for each. In general, initialized sections can be linked into ROM or RAM; uninitialized sections must be linked into RAM. Note that though the .const section may 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 4.1.3, page 4-5, for details). The linker provides MEMORY and SECTIONS directives for performing this process. Refer to subsection 4.1.1, page 4-2, for a complete description of how the compiler uses these 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*.

Sizing the Stack and Heap

The linker provides two options that allow you to specify the size of the .stack and .sysmem sections.

–stack *size* sets the size of the .stack section to *size* words. The value *size* must be constant.

–heap *size* sets the size of the .sysmem section to *size* words. The value *size* must be constant.

The linker always includes the stack section; it includes the .sysmem section only if you use memory allocation functions (such as malloc()). The linker resizes these sections only if the value specified by the option is larger than the input section size (in the standard library, the size is zero, so any –stack or –heap option takes effect). The default size for both sections is 1K (1024) words.

Sample Linker Command File

Example 2–3 shows a typical linker command file that can be used to link a C program. The command file in this example is named link.cmd.

Example 2–3.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
}
```

First, this command file lists several linker options:

–c tells the linker to use the ROM model of autoinitialization.

–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.

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 runtime-support library rts<xx>.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 may 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 3

TMS320C2x/C2xx/C5x C Language

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

ANSI C supersedes the de facto C standard, which was described in the first edition of *The C Programming Language*, by Kernighan and Ritchie. The second edition of *The C Programming Language* is based on the ANSI standard and is a reference. ANSI C encompasses many of the language extensions provided by recent C compilers and formalizes many previously unspecified characteristics of the language.

The TMS320C2x/C2xx/C5x C compiler follows the ANSI C standard. The ANSI standard identifies certain implementation-defined features that may differ from compiler to compiler, depending on the type of processor, the runtime environment, and the host environment. This chapter describes how these and other features are implemented for the TMS320C2x/C2xx/C5x C compiler.

These are the topics covered in this chapter:

Topic **Page**

3.1 Characteristics of TMS320C2x/C2xx/C5x C

The ANSI standard identifies certain features of the C language. These features are affected by characteristics of the target processor, runtime environment, or host environment, which, for reasons of efficiency or practicality, may differ among standard compilers. This section describes how these features 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 the formal ANSI standard and to the *The C Programming Language* (first edition) by Kernighan and Ritchie (K&R).

3.1.1 Identifiers and Constants

- The first 31 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, in all TMS320C2x/C2xx/C5x tools. (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)

3.1.2 Data Types

- For information about the representation of data types, refer to Section 3.2. (ANSI 3.1.2.5, K&R A4.2)
- The type size_t, which is assigned to the result of the sizeof operator, is equivalent to unsigned int. (ANSI 3.3.3.4, K&R A7.4.8)
- The type ptrdiff_t, which is assigned to the result of pointer subtraction, is equivalent to int. (ANSI 3.3.6, K&R A7.7)

3.1.3 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)

3.1.4 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. 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)

3.1.5 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 of type 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)

3.1.6 Preprocessor

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

3.2 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 3–1 lists the size, representation, and range of each scalar data type:

Table 3–1. TMS320C2x/C2xx/C5x C Data Types

			Range	
Type	**Size**	**Representation**	Minimum	Maximum
char, signed char	16 bits	ASCII	–32768	32767
unsigned char	16 bits	ASCII	0	65535
short	16 bits	2s complement	–32768	32767
unsigned short	16 bits	binary	0	65535
int, signed int	16 bits	2s complement	–32768	32767
unsigned int	16 bits	binary	0	65535
long, signed long	32 bits	2s complement	–2147483648	214783647
unsigned long	32 bits	binary	0	4294967295
enum	16 bits	2s complement	–32768	32767
float	32 bits	TMS320C2x/C2xx/C5x	1.19209290e–38	3.4028235e+38
double	32 bits	TMS320C2x/C2xx/C5x	1.19209290e–38	3.4028235e+38
long double	32 bits	TMS320C2x/C2xx/C5x	1.19209290e–38	3.4028235e+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, refer to subsection 5.2.3 on page 5-6.

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).

3.3 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 runtime 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.

3.4 The asm Statement

The TMS320C2x/C2xx/C5x C compiler allows you to imbed TMS320C2x/C2xx/C5x assembly language instructions or directives directly into the assembly language output of the compiler. This capability is provided through an extension to the C language: the *asm* statement. 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 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 *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, it will be detected by the assembler. For more information, refer to the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

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

Note: Avoid Disrupting the C Environment With asm Statements

Be extremely 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 (except where such statements are totally unreachable), it can significantly rearrange the code order near them, possibly causing undesired results. The asm command is provided so that you can access hardware features, which, by definition, C is unable to access.

3.5 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++;
```

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 need to consider very 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 poorer code.

You also need to consider carefully 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.

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 the possibility of corruption, 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 will be 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.

Disabling the Compiler From Using AR6 and AR7

The –r*register* command-line option for the dspcl shell 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 runtime-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 runtime-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
```

3.6 Initializing Static and Global Variables

The ANSI C standard specifies that both static and global (extern) variables without explicit initializations must be preinitialized 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 does not for preinitialize variables; therefore, 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. 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 may have the unwanted effect of significantly increasing the size of the output file.

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.

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.

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. *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 */
```

3.7 Compatibility with K&R C

The ANSI C language is a superset of the de facto C standard defined in the first edition of *The C Programming Language* (K&R). Most programs written for other non-ANSI compilers should correctly compile and run without modification.

However, there are subtle changes in the language that may affect existing code. Appendix C in *The C Programming Language* (second edition) summarizes the differences between ANSI C and the previous C standard (first edition, herein referred to as K&R C).

To simplify the process of compiling existing C programs with the TMS320C2x/C2xx/C5x 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 ANSI C and K&R C are as follows:

- ANSI prohibits two pointers to different types from being combined 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. –pe, which converts code-E errors to warnings, can be used 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 used, error
                       if not */
```

- ANSI specifies that bit fields must be of type integer or unsigned. With –pk, bit fields can be legally declared 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 */
```

3.8 Compiler Limits

Due to the variety of host systems supported by the TMS320C2x/C2xx/C5x C compiler and the limitations of some of these systems, the compiler may not be able to successfully compile source files that are excessively large or complex. Most of these conditions occur during the first compilation pass (parsing). When such a condition occurs, the parser issues a code-I diagnostic message indicating the condition that caused the failure; usually, the message also specifies the maximum value for whatever limit has been 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. The only way to work around exceeding a compiler limit is to simplify the program or parse and preprocess in separate steps.

Many compiler tables have no absolute limits but rather are limited only by the amount of memory available on the host system. Table 3–2 specifies the limits that are absolute. When two values are listed, the first is for PC (MS-DOS or OS/2) host systems, and the second is for other hosts. All the absolute limits equal or exceed those mandated 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:

- ❑ Don’t 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 3–2. 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 initializaion (approximately)
Local initializers	150 levels (approximately)
Nesting of if statements, switch statements, and loops	1500 per initializaion (approximately)
Global symbols	2000 PCs 10000 All others (See Note 5)
Block scope symbols visible at any point	500 PCs 1000 All others
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 4

Runtime Environment

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

Topics in this chapter include:

Topic **Page**

4.1 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 (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. Each block of code or data could be allocated individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although physical memory locations can be accessed with C pointer types).

4.1.1 Sections

The compiler produces seven relocatable blocks of code and data; these blocks, called *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, please read Chapter 2 of the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

There are two basic types of sections:

- **Initialized sections** contain data tables or executable code. The C compiler creates four initialized sections: .text, .cinit, .const, and .switch.
  - The **.text section** is an initialized section that contains all the executable code as well as floating-point constants.
  - The **.cinit section** is an initialized section that contains tables for initializing variables and constants.
  - The **.const section** is an initialized section that contains string constants, and the declaration and initialization of global and static variables (qualified by *const*) that are explicitly initialized.
  - The **.switch section** is an initialized section that contains tables for switch statements.

- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at runtime for creating and storing variables. The compiler creates three uninitialized sections: .bss, .stack, and .sysmem.
  - The **.bss section** is an uninitialized section that reserves space for global and static variables. At program startup time, the C boot routine copies data out of the .cinit section (which may be in ROM) and stores it in the .bss section.
  - The **.stack section** is an uninitialized section used for the C system stack. This memory is used to pass arguments to functions and to allocate space for local variables.
  - The **.sysmem section** is a uninitialized section that reserves space for dynamic memory allocation. The reserved space is utilized by malloc functions. If no malloc functions are used, the size of the section remains 0.

The *assembler* creates an additional section called .data; the C compiler does not use this section.

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

4.1.2 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 runtime 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 runtime stack, be sure to use these registers correctly. (For more information about using these registers, see Section 4.2, page 4-8; for more information about the stack, see Section 4.3, page 4-13.)

The stack size is set by the linker. The linker also creates a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the stack 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 on the linker command line and specifying the size of the stack as a constant immediately after the option.

At system initialization, the SP is set to a designated address for the bottom-of-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 position 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 runtime. A stack overflow will disrupt the runtime environment, causing your program to fail. Be sure to allow enough space for the stack to grow.

4.1.3 Allocating .const to Program Memory

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

Modify the boot routine:

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
```

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 may 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.

4.1.4 Dynamic Memory Allocation

The runtime-support library supplied with the compiler contains several functions (such as malloc, calloc, and realloc) that allow you to dynamically allocate memory for variables at runtime. This is accomplished by declaring a large memory pool, or heap, and then using the functions to allocate memory from the heap. Dynamic allocation is not a standard part of the C language; it is provided by standard runtime-support functions.

This memory pool, or heap, is created by the linker. 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 heap size is 1K words. You can change the size of the memory pool at link time with the –heap option. Specify the size of the memory pool as a constant after the –heap option on the linker command line. The maximum heap size is 32K words.

4.1.5 RAM and ROM Models

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 then 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 runtime). You can specify this *to the linker* by using the –cr linker option. For more information on the –cr linker option, see page 2-52; for more information on system initialization, see Section 4.8, page 4-28.

4.1.6 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. (It reserves space for them in .bss.) Variables declared in the same module are allocated into a single, contiguous block of memory.

4.1.7 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; the least significant bits of the structure word are filled first.

4.1.8 Character String Constants

In C, a character string constant can be used in one of two ways:

- It can 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 is a separate initializer. For more information about initialization, refer to Section 4.8, page 4-28.

- It can be used 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 .byte assembler directive, along with a unique label that points to the string (the terminating 0 byte is also included). In the following example, the label SL5 points to the string from the example above:

  ```
        .const
  SL5   .byte ”abc”, 0
  ```

  String labels have the form **SL**n, n being a number assigned by the compiler to make the label unique. These numbers begin at 0 with an increase of 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 SLn 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 will not be 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:

  ```
  char  *a = ”abc”
  a[1] = ’x’;        /* Incorrect!   */
  ```

4.2 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, it is important that you understand 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 responsiblity 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 4–1. 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

4.2.1 Status Register Fields

Table 4–2 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 the compiler does not expect a particular value. The modified column indicates whether code generated by the compiler ever modifies this field.

Table 4–2. 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

4.2.2 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) will 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 4.3.4, page 4-16, for information on how the LVP is used to access objects on the frame.

4.2.3 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.

4.2.4 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 registers variables is different depending on whether you use the optimizer.

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 registers variables. AR6 is allocated to the first variable, and AR7 is allocated to the second.

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

Setting up a register variable at runtime 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.

Register Variables When the Optimizer is Used

When the optimizer is used, all user register declarations are ignored. The optimizer makes all the decisions on what variables or compiler temporaries are allocated to registers. The optimizer will allocate the variables, not their addresses, directly to registers. The optimizer may allocate the registers AR5, AR6, and AR7 for register variables. Because AR5 is not preserved across function calls, it will not be 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 *Disabling the Compiler From Using AR6 and AR7*, page 3-9, for more information.

4.2.5 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 ever having to save and restore expression registers.

4.2.6 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.

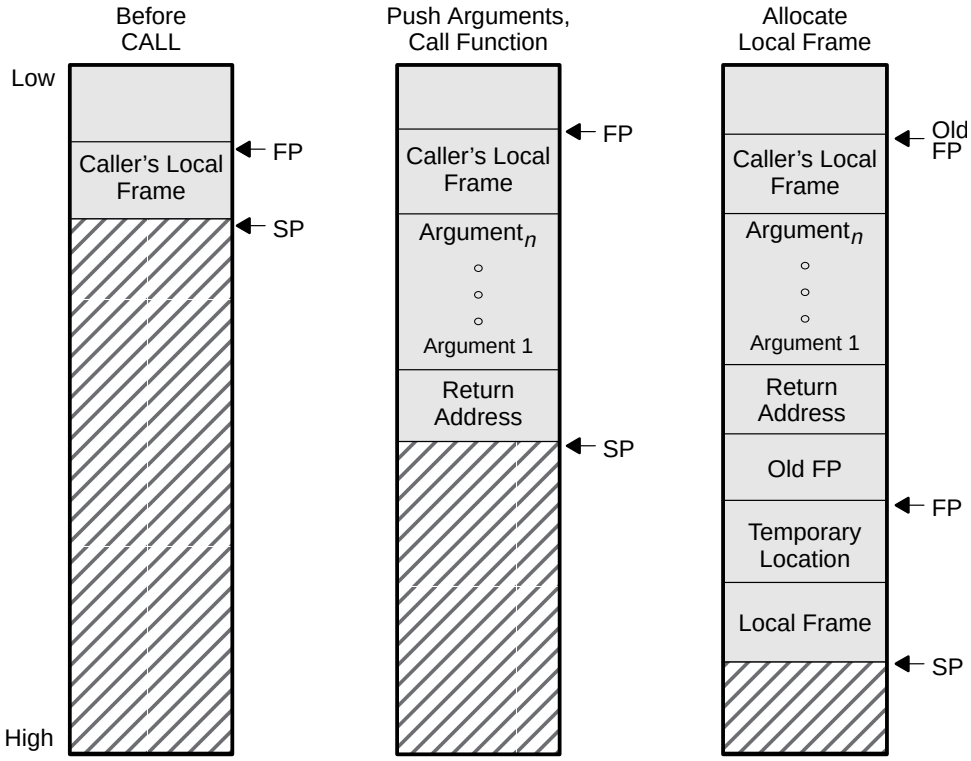
Sixteen-bit types (char, short, int, or pointer) are loaded into the accumulator with correct sign extension.

4.3 Function Calling Conventions

The C compiler imposes a strict set of rules on function calls. Except for special runtime-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 4–1 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 4–1. Stack Use During a Function Call



4.3.1 Function Call

A function performs the following tasks when it calls another function.

1) The ARP must be set to AR1.
2) 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.
3) The caller calls the function.
4) The caller presumes that upon return from the function, the ARP will be set to AR1.
5) 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)

4.3.2 Responsibilities of a Called Function

A called 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 4–1 is an example of TMS320C2x code that performs these tasks.

Example 4–1. 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
```

4.3.3 Special Cases for a Called Function

Returning a structure

If the function returns a structure, the caller allocates space for the structure and then passes the address of the return space to the called 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 be smart about telling 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).

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 runtime-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 subsection 4.3.2, page 4-14, are omitted.

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 subsection 4.3.2, page 4-14, 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.

Using the TMS320C5x RETD *instruction*

The debugger expects the compiler to generate the frame as described above. But when generating code for the 'C5x that will not be run under the debugger, the compiler switches steps 2 and 3 of subsection 4.3.2, page 4-14, as well as steps 11 and 12. The reason is that 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:

```
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.

4.3.4 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.

4.4 Interfacing C With Assembly Language

There are three ways to use assembly language in conjunction with C code:

- Use separate modules of assembled code and link them with compiled C modules (refer to subsection 4.4.1). This is the most versatile method.
- Use inline assembly language, embedded directly in the C source (refer to subsection 4.4.3, page 4-21).
- Modify the assembly language code that the compiler produces (refer to subsection 4.4.4, page 4-22).

4.4.1 Assembly Language Modules

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in Section 4.3 and the register conventions defined in Section 4.2. 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 conventions outlined in Section 4.2, page 4-8.
- You must preserve any dedicated registers modified by a function. Dedicated registers include:

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

  If the SP is used normally, it does not need to be explicitly preserved. The assembly function is free to use the stack as long as anything that is pushed on the stack is popped back off before the functions returns (thus preserving the SP).

  Any register that is not dedicated can be freely used without being preserved.

  For the TMS320C5x only: if your assembly routine does not change the NDX bit of status register PMST from 0 to 1, then the INDX register shadows AR0; thus, preserving AR0 will preserve 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 4–2, page 4-10, 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 about interrupt handling, see Section 4.5, page 4-23.)
- When calling a C function from assembly language, push any arguments onto the stack in reverse order. Pop them off after calling the function.
- When calling 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 language module should use the .cinit section for any purpose other than autoinitialization of global variables. The C startup routine in boot.asm assumes that the .cinit section consists *entirely* of initialization tables. Disrupting the tables by putting other information in .cinit will cause unpredictable results.
- The compiler appends an underscore (_) to the beginning of all identifiers. In assembly language modules, you must use the prefix _ for all objects that are to be accessible from C. For example, a C object named x is called _x in assembly language. For identifiers that are to be used only in an assembly language module or modules, any name that does not begin with an underscore may be safely used without conflicting with a C identifier.
- Any object or function declared in assembly language that is to be accessed or called from C must be declared with the .global directive in the assembler. This defines the symbol as external and allows the linker to resolve references to it.

  Similarly, to access a C function or object from assembly language, declare the C object with .global. This creates an undefined external reference that the linker will resolve.

Example 4–2 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 4–2.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 4–2, the extern declaration of asmfunc is optional, since the function returns an int. Like C functions, assembly functions need be declared only if they return noninteger values. In the assembly language code in Example 4–2, note the underscores on all the C symbol names used in the assembly code.

Further, in Example 4–2 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 done.

4.4.2 How to Define Variables in Assembly Language

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

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

Example 4–3 shows an example of accessing a variable defined in .bss.

Example 4–3.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. For example, a common situation is a lookup table defined in assembly language that you don't want to put in RAM. In this case, you must define a pointer to the object and access it indirectly from C.

The first step is to 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 4–4 shows an example that accesses a variable that is not defined in .bss.

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

(a) C 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
```

(b) Assembly language program

```
extern float sine[];  /* This is the object         */
f = sine[4];          /* Access sine as normal array*/
```

4.4.3 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 is provided so you can access features of the hardware that would be otherwise inaccessible from C. When you use the asm statement, be extremely 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; however, you can 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; simply start the assembly code string with an asterisk (*) as shown below:

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

4.4.4 Modifying Compiler Output

You can inspect and change the assembly language output that the compiler produces 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, refer to Section 2.5, page 2-36.) The warnings in subsection 4.4.3 about disrupting the C environment also apply to modification of compiler output.

4.5 Interrupt Handling

As long as you follow the guidelines in this section, C code can be interrupted and returned to without disrupting the C environment. When the C environment is initialized, the startup routine does not enable or disable interrupts. (If the system is initialized via a hardware reset, interrupts are disabled.) If your system uses interrupts, it is your responsibility to handle any required enabling or masking of interrupts. Such operations have no affect on the C environment and can be easily incorporated with asm statements.

4.5.1 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 runtime-support function I$$SAVE is called to save the complete context of the interrupted function. All registers are saved. Upon return from the interrupt routine, the runtime-support function I$$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.

4.5.2 Using C Interrupt Routines

Interrupts can be handled *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, keep the following points in mind:

- An interrupt routine must be of type void, and it cannot have any arguments.
- The compiler does *not* save all the device registers; the compiler saves only those registers listed in Table 4–1, page 4-9.
- It is your responsibility to 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 may be compiled with the shell option –oe (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 would negate their use.

4.5.3 Using Assembly Language Interrupt Routines

Interrupts can be handled with assembly language code as long as register conventions are followed. 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 4–1, page 4-9, and status bits from Table 4–2, page 4-10, that it modifies.
- If the interrupt routine calls a C function, it must preserve *all* registers listed in Table 4–1 that are not preserved by a call. Any other register may be modified by the C routine.
- Remember to precede the symbol name with an underscore. For example, refer to c_int0 as _c_int0.

4.5.4 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 re-enable 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 may take advantage of the shadow register capability by modifying an assembly time flag in the source of the I$$SAVE/I$$RESTORE routines 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
```

4.6 Integer Expression Analysis

This section describes some special considerations that you should keep in mind when evaluating integer expressions.

4.6.1 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.

4.6.2 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 runtime-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.

4.6.3 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

4.6.4 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:

```
int m1, m2;
int result;
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.

4.7 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 runtime-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 runtime 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.

4.8 System Initialization

Before you can run a C program, the C runtime environment must be created. This task is performed by the C boot routine, which is a function called c_int0. The runtime-support source library contains the source to this routine in a module called boot.asm.

The c_int0 function can be called by reset hardware to begin running the system. The function is in the runtime-support library and must be linked with the C object modules. This occurs automatically when you use the –c or –cr option in the linker and include the library as a linker input file. When C programs are linked, the linker sets the entry point value in the executable output module to the symbol _c_int0.

The c_int0 function performs the following tasks in order to initialize the C environment:

- Creates the section .stack for the runtime stack and sets up the initial stack pointer and frame pointer
- Initializes the fields in the status registers as described in Table 4–2, page 4-10
- Autoinitializes global variables by copying the data from the initialization tables in .cinit to the storage allocated for the variables in .bss. In the RAM autoinitialization model, a loader performs this step before the program runs (it is not performed by the boot routine)
- Calls the function main to begin running the C program

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the four operations listed above in order to correctly to initialize the C environment.

4.8.1 Runtime Stack

The runtime 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 doesn't check to see if the runtime 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.

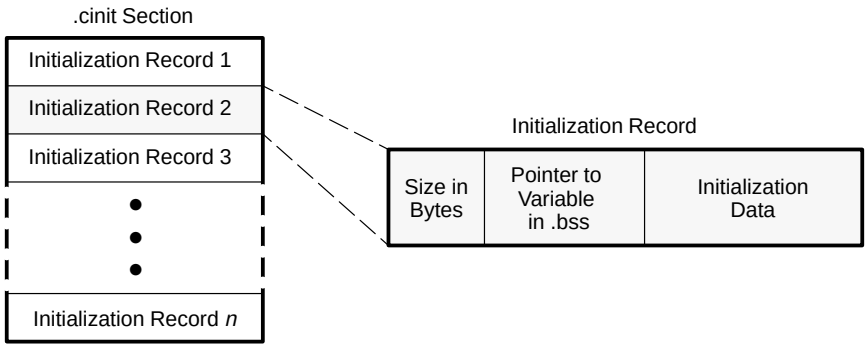
The stack size can be changed at link time by using the –stack option on the linker command line and specifying the stack size as a constant directly after the option.

4.8.2 Autoinitialization of Variables and Constants

Before program execution, any global variables declared as preinitialized must be initialized by the boot function. The compiler builds tables that contain data for initializing global and static variables in a .cinit section in each file. All compiled modules contain these initialization tables. The linker combines them into a single table, which is then used to initialize all the system variables. (Do not place any other data in the .cinit section; this corrupts the tables.)

The tables in .cinit consist of variable-size initialization records. Figure 4–2 shows the format of the .cinit section and the initialization records.

Figure 4–2. Format of Initialization Records in the .cinit Section



The fields of an initialization record contain the following information:

- The first field (word 0) is the size (in words) of the initialization data for the variable.
- The second field (word 1) is the starting address of the area in the .bss section into which the initialization data must be copied. (This field points to a variable's space in .bss.)
- The third field (words 2 through *n*) contains the data that is copied into the variable to initialize it.

The .cinit section contains an initialization record for each variable that must be autoinitialized. 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 would 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 links together 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.

Note that const qualified variables are initialized differently; refer to page 3-10.

Initializing Variables in the RAM Model

The RAM model, specified with the –cr linker option, allows variables to be initialized at *load time* instead of at runtime. This can enhance performance by reducing boot time by saving the memory used by the initialization tables. The RAM option requires the use of a smart loader to perform the initialization as it copies the program from the object file into memory.

In the RAM model, the linker marks the .cinit section with a special attribute. This means that the section is *not* loaded into memory and does *not* occupy space in the memory map. The linker also sets the symbol cinit to –1 to indicate to the C boot routine that the initialization tables are not present in memory; accordingly, no runtime initialization is performed at boot time.

Instead, when the program is loaded into memory, the loader must detect the presence of the .cinit section and its special attribute. Instead of loading the section into memory, the loader uses the initialization tables directly from the object file to initialize the variables in .bss. To use the RAM model, the loader must understand the format of the initialization tables so that it can use them.

A loader is *not* part of the compiler package.

Initializing Variables in the ROM Model

The ROM model is the default model for autoinitialization. To use the ROM model, invoke the linker with the –c option.

Under this method, the .cinit section is loaded into memory (possibly ROM) along with all the other sections, and global variables are initialized at *runtime*. The linker defines a special symbol called cinit that points to the beginning of the 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 .bss. This allows initialization data to be stored in ROM and then copied to RAM each time the program is started.

Chapter 5

Runtime-Support Functions

Some of the tasks that a C program must perform (such as memory allocation, string conversion, and string searches) are not part of the C language. The runtime-support functions, which are included with the C compiler, are standard ANSI functions that perform these tasks. The runtime-support library, rts.src, contains the source for these functions as well as for other functions and routines. All of the ANSI functions except those that require an underlying operating system (such as I/O and signals) are provided. If you use any of the runtime-support functions, be sure to build the appropriate library, according to the device, using the library-build utility (see Chapter 6); then include that library as linker input when you link your C program.

These are the topics covered in this chapter:

Topic **Page**

5.1 Libraries

Four libraries are included with the TMS320C2x/C2xx/C5x C compiler: three object libraries containing object code for the runtime support and one library containing source code for the functions in the object libraries.

- *rts25.lib, rts2xx.lib,* and *rts50.lib* contain the ANSI runtime-support object libraries.
- *rts.src* contains the source for the ANSI runtime-support routines.

The object libraries include the standard C runtime-support functions described in this chapter, the floating-point routines, and the system startup routine, _c_int0. The object libraries are built from the C and assembly source contained in rts.src.

When you link your program, you must specify an object library as one of the linker input files so that references to runtime-support functions can be resolved. You should usually specify libraries *last* on the linker command line because the assembler searches for unresolved references when it encounters a library on the command line. When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, refer to Section 2.8 on page 2-48.

There is one header file, values.h, in rts.src. It is not a standard header. It is provided so that you can customize the functions. It contains definitions necessary for recompiling the trigonometric and transcendental math functions.

5.1.1 Modifying a Library Function

You can inspect or modify library functions by using the archiver to extract the appropriate source file or files from rts.src. 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, refer to Chapter 7 of the *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*.

5.1.2 Building a Library With Different Options

You can create a new library from rts.src by using the runtime-support installation utility, dspmk. For example, use this command to build a fast, optimized runtime-support library:

```
dspmk –-u –o2 –x –mf 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 –o2 and –x options are the optimizer options used to build the rts.lib supplied with the toolset. The –mf option tells the compiler to ignore code length and generate the fastest possible code.

5.2 Header Files

Each runtime-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 runtime-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 runtime-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.

Subsections 5.2.1 through 5.2.12 describe the header files that are included with the C compiler. Section 5.3, page 5-13, lists the functions that these headers declare.

5.2.1 Diagnostic Messages (assert.h)

The *assert.h* header defines the assert macro, which inserts diagnostic failure messages into programs at runtime. The assert macro tests a runtime 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 (via 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 does nothing. If NDEBUG *is not* defined, then assert is enabled.

The assert macro is defined as follows:

```
#ifdef NDEBUG
#define assert(ignore) ((void)0)
#else
#define assert(expr) ((void)((_expr) ? 0 :
       (printf(”Assertion failed, (”#_expr”), file %s,   \
       line %d\n, __FILE__, __LINE__),                   \
       abort () )))
#endif
```

5.2.2 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).

The character conversion functions convert characters to lower case, upper case, or ASCII and return the converted character.

Character-typing functions have names in the form **is***xxx* (for example, *isdigit*). 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. However, 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*). If an argument passed to one of these macros has side effects, the function version should be used instead. For example, _isdigit (*p++) has the effect of incrementing p. The macro may increment p more than once, but the function _isdigit will assure the correct result.

5.2.3 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 5–1 and Table 5–2 list these macros and the limits with which they are associated.

Table 5–1. 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	–32768	Minimum value for a signed char
SCHAR_MAX	32767	Maximum value for a signed char
UCHAR_MAX	65535	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	–32768	Minimum value for a short int
SHRT_MAX	32767	Maximum value for a short int
USHRT_MAX	65535	Maximum value for an unsigned short int
INT_MIN	–32768	Minimum value for an int
INT_MAX	32767	Maximum value for an int
UINT_MAX	65535	Maximum value for an unsigned int
LONG_MIN	–2147483648	Minimum value for a long int
LONG_MAX	2147483647	Maximum value for a long int
ULONG_MAX	4294967295	Maximum value for an unsigned long int

Table 5–2. 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.

5.2.4 Floating-Point Math (math.h)

The *math.h* header defines several trigonometric, exponential, and hyperbolic math functions. These math functions expect double-precision floating-point arguments and return double-precision floating-point values.

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.

5.2.5 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.

5.2.6 Variable Arguments (stdarg.h)

Some functions can have a variable number of arguments whose types can differ; such a function is called a *variable-argument function*. The *stdarg.h* header declares three macros and a type that help you to use variable-argument functions:

The three 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 it knows the number and types of arguments actually passed to it.

5.2.7 Standard Definitions (stddef.h)

The *stddef.h* header defines two types and two macros. The types include:

- *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 include:

- The *NULL* macro, which expands to a null pointer constant(0)
- The *offsetof(type, identifier)* macro, which 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 runtime-support functions.

5.2.8 General Utilities (stdlib.h)

The *stdlib.h* header declares several functions, one macro, and two types. The types include:

- *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 stdlib.h header macro, *RAND_MAX*, is the maximum random number the rand function will return.

The header also declares many of the common library functions:

- **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 and specifying the desired heap size as a constant directly after the 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

5.2.9 String Functions (string.h)

The *string.h* header declares standard functions that allow you to 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 have names such as **mem***xxx*.

When you use functions that move or copy strings, be sure that the destination is large enough to contain the result.

5.2.10 Time Functions (time.h)

The *time.h* header declares one macro, several types, and 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 savings 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 header declares three types:

- *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 Savings Time flag        */
```

tm_isdst can have one of three values:

- A *positive* value if Daylight Savings Time is in effect
- *Zero* if Daylight Savings Time is not in effect
- A *negative* value if the information is not available

Note: Customizing Time Functions

All of the time functions depend on the clock and time functions, which you must customize for your system.

5.2.11 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:

- The *inport* macro, which reads a value from the specified port and returns the value via the pointer *ret*
- The *outport* macro, which writes a value to the specified port and has no return value

The functions include:

- The *_inport<x>* functions, which access the port of the same number and return that value as an int. These functions follow this syntax:

  _inport<x>( ) where $0 \leq x \leq 15$

- The *_outport<x>* functions, which access the port of the same number and have no return value. These functions follow this syntax:

  _outport<x>(int value) where $0 \leq x \leq 15$

- The *_in_port* function, which reads an int from the specified port and returns the value. This function follows this syntax:

  _in_port (int port)

- The *_out_port* function, which 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 will remain in user code. Therefore, when the port number is a variable, set _PSWITCH to 1 to call the correct _in_port or _out_port function. The macros will always work, regardless of the value of _PSWITCH.

5.2.12 Bypass Normal Function Call and Return Conventions (setjmp.h)

The *setjmp*.h header defines one type, one macro, and one 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

5.3 Summary of Runtime-Support Functions and Macros

Refer to the following pages for information about functions and macros:

Error Message Macro (assert.h)	**Description**
void **assert**(int expression);‡	Inserts diagnostic messages into programs
Character-Typing Conversion Functions (ctype.h)	**Description**
int **isalnum(**char c); †	Tests c to see if it's an alphanumeric ASCII character
int **isalpha**(char c); †	Tests c to see if it's an alphabetic ASCII character
int **isascii**(char c); †	Tests c to see if it's an ASCII character
int **iscntrl**(char c); †	Tests c to see if it's a control character
int **isdigit**(char c); †	Tests c to see if it's a numeric character
int **isgraph**(char c); †	Tests c to see if it's any printing character except a space
int **islower**(char c); †	Tests c to see if it's a lowercase alphabetic ASCII character
int **isprint**(char c); †	Tests c to see if it's a printable ASCII character (including spaces)
int **ispunct**(char c); †	Tests c to see if it's an ASCII punctuation character
int **isspace**(char c); †	Tests c to see if it's an ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, or newline characters
int **isupper**(char c); †	Tests c to see if it's an uppercase alphabetic ASCII character
int **isxdigit**(char c); †	Tests c to see if it's a hexadecimal digit
int **toascii**(char c); †	Masks c into a legal ASCII value
int **tolower**(int char c); †	Converts c to lowercase if it's uppercase
int **toupper**(int char c); †	Converts c to uppercase if it's lowercase
Floating-Point Math Functions (math.h)	**Description**
double **acos**(double x);	Returns the arc cosine of x
double **asin**(double x);	Returns the arc sine of x
double **atan**(double x);	Returns the arc tangent of x
double **atan2**(double y, double x);	Returns the inverse tangent of y/x
double **ceil**(double x);†	Returns the smallest integer greater than or equal to x
double **cos**(double x);	Returns the cosine of x
double **cosh**(double x);	Returns the hyperbolic cosine of x

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

Floating-Point Math Functions (continued)	**Description**
double **exp**(double x); §	Returns the exponential function of x
double **fabs**(double x);	Returns the absolute value of x
double **floor**(double x); †	Returns the largest integer less than or equal to x
double **fmod**(double x, double y); †	Returns the floating-point remainder of x/y
double **frexp**(double value, int *exp);	Breaks value into a normalized fraction and an integer power of 2
double **ldexp**(double x, int exp);	Multiplies x by an integer power of 2
double **log**(double x);	Returns the natural logarithm of x
double **log10**(double x);	Returns the base-10 (common) logarithm of x
double **modf**(double value, int *iptr);	Breaks value into into a signed integer and a signed fraction
double **pow**(double x, double y);	Returns x raised to the power y
double **sin**(double x);	Returns the sine of x
double **sinh**(double x);	Returns the hyperbolic sine of x
double **sqrt**(double x);	Returns the nonnegative square root of x
double **tan**(double x);	Returns the tangent of x
double **tanh**(double x);	Returns the hyperbolic tangent of x
Variable-Argument Functions and Macros (stdarg.h)	**Description**
type **va_arg**(va_list ap); ‡	Accesses the next argument of type *type* in a variable-argument list
void **va_end**(va_list ap); ‡	Resets the calling mechanism after using va_arg
void **va_start**(va_list ap); ‡	Initializes ap to point to the first operand in the variable-argument list
Nonlocal Jumps Macro and Function (setjmp.h)	**Description**
int **setjmp**(jmp_buf env); ‡	Saves calling environment for later use by longjmp function
void **longjmp**(jmp_buf env, int returnval);	Uses jmp_buf argument to restore a previously saved program environment

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

General Utilities (stdlib.h)	Description
int **abs**(int j); §	Returns the absolute value of j
void **abort**(void)	Terminates a program abnormally
void **atexit**(void (*fun)(void));	Registers the function pointed to by fun, to be called without arguments at normal program termination
double **atof**(const char *nptr);	Converts a string to a floating-point value
int **atoi**(const char *nptr);	Converts a string to an integer value
long **atol**(const char *nptr);	Converts a string to a long integer value
void ***bsearch**(const void *key, 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
void ***calloc** (size_t nmemb, size_t size);	Allocates and clears memory for n objects, each of size bytes
div_t **div**(int numer, int denom);	Divides numer by denom
void **exit**(int status);	Terminates a program normally
void **free**(void *ptr);	Deallocates memory space allocated by malloc, calloc, or realloc
long **labs**(long j); §	Returns the absolute value of j
ldiv_t **ldiv** (long numer, long denom);	Divides numer by denom
int **ltoa**(long n, char *buffer);	Converts n to the equivalent string
void ***malloc**(size_t size);	Allocates memory for an object of size bytes
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc
void **qsort**(void *_base, size_t nmemb, size_t _size, int (*_compar) (void *, void *));	Sorts an array of n members; base points to the first member of the unsorted array, and size specifies the size of each member
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX
void ***realloc**(void *ptr, size_t size);	Changes the size of an allocated memory space
void **srand**(unsigned int seed);	Resets the random number generator
double **strtod**(const char *nptr, char **endptr);	Converts a string to a floating-point value
long **strtol**(const char *nptr, char **endptr, int base);	Converts a string to a long integer
unsigned long **strtoul**(const char *nptr, char **endptr**,** int base**);**	Converts a string to an unsigned long integer

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

String Functions (string.h)	Description
void ***memchr**(void *s, int c, size_t n); †	Finds the first occurrence of c in the first n characters of s
int **memcmp**(void *s1, void *s2, size_t n); †	Compares the first n characters of s1 to s2
void ***memcpy**(void *s1, void *s2, size_t n); †	Copies n characters from s2 to s1
void ***memmove**(void *s1, void *s2, size_t n);	Moves n characters from s2 to s1
void ***memset**(void *s, int c, size_t n); †	Copies the value of c into the first n characters of s
char ***strcat**(char *s1, char *s2); †	Appends s2 to the end of s1
char ***strchr**(char *s, int c); †	Finds the first occurrence of character c in s
int **strcmp**(char *s1, char *s2); †	Compares strings and returns one of the following values: <0 if s1 is less than s2; =0 if s1 is equal to s2; >0 if s1 is greater than s2
int ***strcoll**(char *s1, char *s2);	Compares strings and returns one of the following values, depending on the locale: <0 if s1 is less than s2; =0 if s1 is equal to s2; >0 if s1 is greater than s2
char ***strcpy**(char *s1, char *s2); †	Copies string s2 into s1
size_t **strcspn**(char *s1, char *s2);	Returns the length of the initial segment of s1 that is made up entirely of characters that are not in s2
char ***strerror**(int errnum);	Maps the error number in errnum to an error message string
size_t **strlen**(char *s); †	Returns the length of a string
char ***strncat**(char *s1, char *s2, size_t n);	Appends up to n characters from s1 to s2
int **strncmp**(char *s1, char *s2, size_t n);	Compares up to n characters in two strings
char ***strncpy**(char *s1, char *s2, size_t n);	Copies up to n characters of s2 to s1
char ***strpbrk**(char *s1, char *s2);	Locates the first occurrence in s1 of *any* character from s2
char ***strrchr**(char *s1, char c); †	Finds the last occurrence of character c in s
size_t **strspn**(char *s1, char *s2);	Returns the length of the initial segment of s1, which is entirely made up of characters from s2
char ***strstr**(char *s1, char *s2);	Finds the first occurrence of s2 in s1
char ***strtok**(char *s1, char *s2);	Breaks s1 into a series of tokens, each delimited by a character from s2

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

Time Functions (time.h)	**Description**
char ***asctime**(const struct tm *timeptr);	Converts a time to a string
clock_t **clock**(void);	Determines the processor time used
char ***ctime**(const time_t *timeptr);	Converts calendar time to local time
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times
struct tm ***gmtime**(const time_t *timer);	Converts calendar time to Greenwich Mean Time
struct tm ***localtime**(const time_t *timer);	Converts calendar time to local time
time_t **mktime**(struct tm *timeptr);	Converts local time to calendar time
size_t **strftime**(char *s, size_t maxsize, const char *format, const struct tm *timeptr);	Formats a time into a character string
time_t **time**(time_t *timer);	Returns the current calendar time
I/O Port Macros (ioports.h)	**Description**
int **inport**(int port, int *ret); ‡	Returns a value from the specified port via the pointer *ret*
void **outport**(int port, int value); ‡	Writes a value to the specified port and has no return value

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

5.4 Functions Reference

The remainder of this chapter is a reference for all runtime-support functions and macros.

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 2.4, page 2-30.

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. x 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 — *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, see subsection 5.2.10, page 5-10.

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. x 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 *expression***);**

Defined in

assert.h as a macro

Description

The **assert** macro tests an expression; depending upon the value of the expression, assert either aborts execution and issues a message or continues execution. This macro is useful for debugging.

- If expression is false, the assert macro writes information about the call that failed to the standard output, and then aborts execution.
- If expression is true, the assert macro does nothing.

The header file that declares 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 either

- provide your own version of printf, or
- 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 when this code runs, a message such as “Assertion failed (j), file foo.c, line 123” is sent to standard output.

```
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*, *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 — *Exit Without Arguments)*

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 *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
```

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 nptr 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 nptr 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 nptr points to the string; the string must have the following format:

  [space] [sign] digits

The space is one or more of the following characters; 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 then 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 *nmemb***, size_t** *size***);**

Defined in

memory.c in rts.src

Description

The **calloc** function allocates size bytes 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 linker sets the size of this section from the value specified by the –heap option. Default heap size is 1K words. For more information, see subsection 4.1.4, page 4-6.

Example

This example uses the calloc routine to allocate and clear 10 bytes.

```
prt = calloc (10,2);   /*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 return value can be converted to seconds by dividing 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 granularity 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, see subsection 5.2.10, page 5-10.

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 may 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 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 **timptr***);**

Defined in

ctime.c in rts.src

Description

The **ctime** function converts the calendar time (pointed to by timer and represented as a value of type time_t) to 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, see subsection 5.2.10, page 5-10.

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, see subsection 5.2.10, page 5-10.

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 get 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 raised to the power 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 remainder after dividing x by y an integral number of times. 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 **ptr***);*

Defined in

memory.c in rts.src

Description

The **free** function deallocates memory space (pointed to by ptr) 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, refer to subsection 4.1.4, page 4-6.

Example

This example allocates 10 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 and an integer power of 2. The function returns a number x, with a magnitude in the range [1/2,1) or 0, so that value == $x \times 2^{exp}$. The frexp function stores the power 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.

For more information about the functions and types that the time.h header declares, see subsection 5.2.10, page 5-10.

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 will result 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 will 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* (scheduled to be published in the second quarter of 1995), 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 character is the type of character that it was tested to be), 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.

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 a power of 2 given by 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 */
```

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 and represented as a value of type time_t) into a broken-down time in a structure. The function returns a pointer to the structure representing the converted time.

For more information about the functions and types that the time.h header declares, see subsection 5.2.10, page 5-10.

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 (or common 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 */
```

ltoa — *Convert Long Integer to ASCII*

Syntax

#include <stdlib.h>

int ltoa(long *n***, char** **buffer***);*

Defined in

ltoa.c in rts.src

Description

The **ltoa** function converts a long integer n to the equivalent ASCII string and writes it into buffer with a null terminator. If the input number n is negative, a leading minus sign is output. The ltoa function returns the number of characters placed in the buffer, not including the terminator.

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, defined in an uninitialized named section called .sysmem in memory.c. The linker sets the size of this section from the value specified by the –heap option. Default heap size is 1K words. For more information, see subsection 4.1.4, page 4-6.

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 **s***, **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 s 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 **s1***, const void** **s2***, size_t** *n***);**

Defined in

memcmp.c in rts.src

Description

The **memcmp** function compares the first n characters of the object that s2 points to with the object that s1 points to. The function returns one of the following values:

<	**0**	if *s1 is less than *s2
	0	if *s1 is equal to *s2
>	**0**	if *s1 is greater than *s2

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 *s, int c, size_t n);
```

Defined in

memset.c in rts.src

Description

The **memset** function copies the value of c into the first n characters of the object that s points to. The function returns the value of s. 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.

> **Note: Accessing Objects After Calling the minit Function**
>
> 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.

The memory that minit uses is in a special memory pool or heap, defined in an uninitialized named section called .sysmem in memory.c. The linker sets the size of this section from the value specified by the –heap option. Default heap size is 1K words. For more information, see subsection 4.1.4, page 4-6.

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 a time value of type time_t. 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.

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]);
```

For more information about the functions and types that the time.h header declares, see subsection 5.2.10, on page 5-10.

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 may occur 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** *n***, size_t** *size***, int** (***compar*)
(**const void** *, **const void** *));

Defined in

qsort.c in rts.src

Description

The **qsort** function sorts an array of n 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 random number generator 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 *ptr, size_t size);
```

Defined in

memory.c in rts.src

Description

The **realloc** function changes the size of the allocated memory pointed to by ptr 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 ptr is 0, realloc behaves like malloc.
- If ptr points to unallocated space, the 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 ptr is not null, realloc frees the space that ptr points to.

If, in order to allocate more space, the entire object must be moved, 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, defined in an uninitialized named section called .sysmem in memory.c. The linker sets the size of this section from the value specified by the –heap option. Default heap size is 1K words. For more information, see subsection 4.1.4, page 4-6.

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 one type, one macro, and one 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 will 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. x is an angle expressed in radians. An argument with a large magnitude may produce a result with little or no significance.

Example

```
double radian, sval;  /* sval is returned 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

The runtime-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 supplies a minimal version of sprintf() that performs only the formatting required by time(). See the description of ti_sprintf on page 5-58 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 non-negative 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 */
```

strcat — *Concatenate Strings*

Syntax

#include <string.h>

char *strcat(char **s1***, **char** **s2***);**

Defined in

strcat.c in rts.src

Description

The **strcat** function appends a copy of s2 (including the terminating null character) to the end of s1. The initial character of s2 overwrites the null character that originally terminated s1. The function returns the value of s1. The strcat function is expanded inline when the –x option is used.

Example

In the following example, the character strings pointed to by a, b, and c were 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 **s*, **char** *c*);**

Defined in

strchr.c in rts.src

Description

The **strchr** function finds the first occurrence of c in s. 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 **s1*, **const char** **s2*);**
int strcmp(const char **s1*, **const char** **s2*);**

Defined in

strcmp.c in rts.src

Description

The **strcmp** and **strcoll** functions compare s2 with s1. 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 *s1 is less than *s2
	0	if *s1 is equal to *s2
>	**0**	if *s1 is greater than *s2

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 **s1**, **const char** **s2**);**

Defined in

strcpy.c in rts.src

Description

The **strcpy** function copies s2 (including a terminating null character) into s1. If you attempt to copy strings that overlap, the function's behavior is undefined. The function returns a pointer to s1. 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 **s1**, **const char** **s2**);**

Defined in

strcspn.c in rts.src

Description

The **strcspn** function returns the length of the initial segment of s1, which is made up entirely of characters that are not in s2. If the first character in s1 is in s2, 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 **s***, **size_t** *maxsize***, const char** **format*,
const struct tm **timeptr***);*

Defined in

strftime.c in rts.src

Description

The **strftime** function formats a time (pointed to by timeptr) according to a format string and returns the formatted result in the string s. Up to maxsize characters can be written to s. 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, see subsection 5.2.10, page 5-10.

strlen *Find String Length*

Syntax

#include <string.h>

size_t strlen(const char *s**);**

Defined in

strlen.c in rts.src

Description

The **strlen** function returns the length of s. 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 *s1**, const char** *s2**, size_t** *n**);**

Defined in

strncat.c in rts.src

Description

The **strncat** function appends up to n characters of s2 (including a terminating null character) to the end of s1. The initial character of s2 overwrites the null character that originally terminated s1; strncat appends a null character to the result. The function returns the value of s1.

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 **s1***, const char** **s2***, size_t** *n***);**

Defined in

strncmp.c in rts.src

Description

The **strncmp** function compares up to n characters of s2 with s1. The function returns one of the following values:

< 0 if *s1 is less than *s2
0 if *s1 is equal to *s2
> 0 if *s1 is greater than *s2

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 **s1**, **const char** **s2**, **size_t** *n*);

Defined in

strncpy.c in rts.src

Description

The **strncpy** function copies up to n characters from s2 into s1. If s2 is n characters long or longer, the null character that terminates s2 is not copied. If you attempt to copy characters from overlapping strings, the function's behavior is undefined. If s2 is shorter than n characters, strncpy appends null characters to s1 so that s1 contains n characters. The function returns the value of s1.

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 **s1**, **const char** **s2**);**

Defined in

strpbrk.c in rts.src

Description

The **strpbrk** function locates the first occurrence in s1 of any character in s2. 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 **s**, **int** **c**);**

Defined in

strrchr.c in rts.src

Description

The **strrchr** function finds the last occurrence of c in s. 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 **s1**, **const char** **s2**);**

Defined in

strspn.c in rts.src

Description

The **strspn** function returns the length of the initial segment of s1, which is entirely made up of characters in s2. If the first character of s1 is not in s2, 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 **s1**, **const char** **s2**);**

Defined in

strstr.c in rts.src

Description

The **strstr** function finds the first occurrence of s2 in s1 (excluding the terminating null character). If strstr finds the matching string, it returns a pointer to the located string; if it doesn't find the string, it returns a null pointer. If s2 points to a string with length 0, strstr returns s1.

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 *nptr, char **endptr);
long int strtol(const char *nptr, char **endptr, int base);
unsigned long int strtoul(const char *nptr, 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 nptr 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 be specified in the following format:

  [space] [sign] digits [.digits] [e|E [sign] integer]

The space is indicated by one or more of the following characters: space bar, horizontal or vertical tab, carriage return, form feed, or newline. Following the space is an optional sign, and then 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 **s1*, **const char** **s2*);

Defined in

strtok.c in rts.src

Description

Successive calls to the **strtok** function break s1 into a series of tokens, each delimited by a character from s2. Each call returns a pointer to the next token. The first call to strtok uses the string s1. Successive calls use a null pointer as the first argument. The value of s2 can change at each invocation. It is important to note that s1 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. x is an angle 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 as a value of type time_t. The value is the number of 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, see subsection 5.2.10, page 5-10.

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 will be 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 runtime-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, _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, to upper or lower case:

- The **tolower** function converts an uppercase argument to lowercase. If c is not an uppercase letter, tolower returns it unchanged.
- The **toupper** function converts a lowercase argument to uppercase. If c is not a lowercase letter, 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 can be called with a varying number of arguments that have varying types. Such functions, called *variable-argument functions*, can use the following macros to step through argument lists at runtime. 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 rightmost 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 6

Library-Build Utility

When using the TMS320C2x/C2xx/C5x 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 runtime-support libraries, this package includes the source file, rts.src, that contains all runtime-support functions. You can custom build your own runtime-support libraries for your selected options 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*.

You install the dspmk utility on your host according to your system's conventions by using the procedure described in the *TMS320C1x/C2x/C2xx/C5x Code Generation Tools Getting Started*. All tools must be placed in your PATH. The utility ignores and disables the environment variables TMP, C_OPTION, and C_DIR.

These are the topics covered in this chapter:

Topic **Page**

6.1 Invoking the Library-Build Utility

The general syntax for invoking the library utility is:

dspmk [*options*] *src_arch1* [**–l***obj.lib1*] [*src_arch2* [**–l***obj.lib2*]] ...

dspmk is the command that invokes the utility.

options can appear anywhere on the command line or in a linker command file. (Options are discussed in Section 6.2 and below.)

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. An object library cannot be built from multiple source archive files.

Library-Build Utility-Specific 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 instructs dspmk to use header files contained in the source archive and leave them in the current directory after the utility has completed execution. You will probably want to use this option to install the runtime-support header files from the rts.src archive that is shipped with the tools.

––k instructs the dspmk utility to 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 instructs the utility to suppress header information (quiet).

––u instructs dspmk not to 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 some flexibility in modifying runtime-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).

6.2 Options Summary

The other options that can be used with the dspmk utility correspond directly to the options that the compiler uses. These options are described in detail in subsection 2.1.3, page 2-5. The following table provides a summary of the options that you can use with the utility.

General Options	Effect
–g	enable symbolic debugging
–r*register*	reserve global register
–v*xx*	specify target processor TMS320C*xx* (25, 50, 2xx)
Parser Options	**Effect**
–pk	make code K&R compatible
–pw	suppress warning messages
–p?	enable trigraph expansion
Optimizer Options	**Effect**
–o0	compile with optimization; register optimization
–o1	compile with optimization; + local optimization
–o2 (or –o)	compile with optimization; + global optimization
–o3	compile with optimization; + file optimization Note that dspmk automatically sets –oI0 and –op0.
–oe	assume no calls by interrupts
–ox (equivalent to –x2)	define _INLINE + above + invoke optimizer (at –o2 if not specified differently)
Inlining Options	**Effect**
–x1	enable intrinsic function inlining
–x2 (or –x)	define _INLINE + above + invoke optimizer (at –o2 if not specified differently)

Runtime Model Options	Effect
–ma	assume aliased variables
–mb	avoid RPTK for structure moves
–ml	disable LDPK optimization
–mn	enable optimization disabled by –g
–ms	optimize for space instead of for speed
–mx	avoid ’C5x silicon bugs
Type Checking Options	**Effect**
–tf	relax prototype checking
–tp	relax pointer combination checking
Assembler Options	**Effect**
–ap	’C2x to ’C2xx or ’C5x port switch
–app	’C2x to ’C2xx port switch and define .TMS32025 and .TMS3202XX
–as	keep labels as symbols
Default File Extensions	**Effect**
–ea<.*ext*>	extension for assembly files (default is .asm)
–eo<.*ext*>	extension for object files (default is .obj)

Appendix A

Optimization

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. (For details about the –o options, refer to Section 2.3 on page 2-25.) However, the code generator performs some optimizations, particularly the TMS320C2x/C2xx/C5x-specific optimizations, that you cannot selectively enable or disable.

This appendix describes two categories of optimizations: general and TMS320C2x/C2xx/C5x-specific. General optimizations improve any C code, and TMS320C2x/C2xx/C5x-specific optimizations are designed especially for the TMS320C2x/C2xx/C5x architecture. Both kinds of optimizations are performed throughout the compiler.

These are the optimizations covered in this appendix:

TMS320C2x/C2xx/C5x-Specific Optimizations

- Cost-based register allocation
- Autoincrement addressing
- Repeat blocks
- Delays, branches, calls, and returns

General Optimizations

- Algebraic reordering, symbolic simplification, constant folding
- Alias disambiguation
- Copy propagation
- Common subexpression elimination
- Redundant assignment elimination
- Branch optimizations, control-flow simplification
- Loop induction variable optimizations, strength reduction
- Loop rotation
- Loop invariant code motion
- Inline expansion of runtime-support library functions

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 don't overlap may be allocated to the same register.

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 A–1.

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 will allocate an AR as the loop counter and implement the loop with a BANZ instruction. See Example A–1 and Example A–5.

Example A–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.

Delays, branches, 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 A–2.

Example A–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
      . . .
```

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 A–3.

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 I (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.

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 A–3 and Example A–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 A–3 and Example A–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 A–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 A–3.

Example A–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.

Branch Optimizations / 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 A–4.

Example A–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.

Loop Induction Variable Optimizations / 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 A–1 and Example A–5.

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.

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 A–5.

Inline Expansion of Runtime-Support Library Functions

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

Example A–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.)

Appendix B

Glossary

A

ANSI: American National Standards Institute.

absolute address: An address that is permanently assigned to a memory location.

absolute lister: A debugging tool that allows you to create assembler listings that contain absolute addresses.

alignment: A process in which the linker places an output section at an address that falls on an *n*-bit boundary, where *n* is a power of 2. You can specify alignment with the SECTIONS linker directive.

allocation: A process in which the linker calculates the final memory addresses of output sections.

archive library: A collection of individual files that have been grouped into a single file.

archiver: A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro directives. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

assembly-time constant: A symbol that is assigned to a constant value with the .set or .equ directive.

assignment statement: A statement that assigns a value to a variable.

autoinitialization: The process of initializing global C variables (contained in the .cinit section) before beginning program execution.

auxiliary entry: The extra entry that a symbol may have in the symbol table and that contains additional information about the symbol (whether the symbol is a filename, a section name, a function name, etc.).

B

.bss: One of the default COFF sections. You can use the .bss directive to reserve a specified amount of space in the memory map that can later be used for storing data. The .bss section is uninitialized.

binding: A process in which you specify a distinct address for an output section or a symbol.

block: A set of declarations and statements that are grouped together with braces.

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 program that translates C source statements into assembly language source statements.

code generator: A compiler tool that takes the intermediate file produced by the parser or the .opt file produced by the optimizer and produces an assembly language source file.

command file: A file that contains linker options and names input files for the linker.

comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

common object file format (COFF): An object file format that promotes modular programming by supporting the concept of *sections.*

conditional processing: A method of processing one block of source code or an alternate block of source code, according to the evaluation of a specified expression.

configured memory: Memory that the linker has specified for allocation.

constant: A numeric value that can be used as an operand.

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

D

.data: One of the default COFF sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

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: Memory allocation created by several functions (such as malloc, calloc, and realloc) that allows you to dynamically allocate memory for variables at runtime. This is accomplished by declaring a large memory pool, or heap, and using the functions to allocate memory from the heap.

E

emulator: A hardware development system that emulates TMS320C2x, TMS320C2xx, or TMS320C5x operation.

entry point: The starting execution point in target memory.

executable module: An object file that has been linked and can be executed in a 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 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 header: A portion of a COFF object file that contains general information about the object file (such as the number of section headers, the type of system the object file can be downloaded to, the number of symbols in the symbol table, and the symbol table's starting address.)

G

global: A kind of symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.

GROUP: An option of the SECTIONS directive that forces specified output sections to be allocated contiguously (as a group).

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.

hole: An area between the input sections that compose an output section that contains no actual code or data.

I

incremental linking: The linking of files that have already been linked.

initialized section: A COFF section that contains executable code or initialized data. An initialized section can be built up with the .data, .text, or .sect directive.

input section: A section from an object file that will be linked into an executable module.

integrated preprocessor: The C preprocessor is integrated with the parser, allowing for faster compilation. Standalone preprocessing or a preprocessed listing is also available.

interlist utility: This utility interlists your original C source file with the assembly language output from the assembler.

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.

line number entry: An entry in a COFF output module that maps lines of assembly code back to the original C source file that created them.

linker: A software tool that combines object files to form an object module that can be allocated into system memory and executed by the device.

listing file: An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the SPC.

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 source statements that are substituted for the macro call and are subsequently assembled.

macro library: An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of .asm.

magic number: A COFF file header entry that identifies an object file as a module that can be executed by the TMS320C2x, TMS320C2xx, or TMS320C5x devices.

map file: An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which they were defined.

memory map: A map of target system memory space, which is partitioned off into functional blocks.

mnemonic: An instruction name that the assembler translates into machine code.

model statement: Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.

N

named section: An initialized section that is defined with a .sect directive, or an uninitialized section that is defined with a .usect directive.

O

object file: A file that has been assembled or linked and contains machine-language object code.

object library: An archive library made up of individual object files.

operand: The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.

optional header: A portion of a COFF object file that the linker uses to perform relocation at download time.

options: Command parameters that allow you to request additional or specific functions when you invoke a software tool.

optimizer: A software tool that improves the execution speed and reduces the size of C programs.

output module: A linked, executable object file that can be downloaded and executed on a target system.

output section: A final, allocated section in a linked, executable module.

overlay pages: Multiple areas of physical memory that occupy the same address space at different times. TMS320C2x, TMS320C2xx, and TMS320C5x devices can map different pages into the same address space in response to hardware select signals.

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.

preprocessor: A software tool that handles macro definitions and expansions, included files, conditional compilation, and preprocessor directives.

R

RAM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the –cr option. The RAM model allows variables to be initialized at load time instead of runtime.

raw data: Executable code or initialized data in an output section.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

ROM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the –c option. In the ROM model, the linker loads the .cinit section of data tables into memory, and variables are initialized at runtime.

runtime environment: Memory and register conventions, stack organization, function call conventions, and system initialization; also information on interfacing assembly language with C programs.

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 occupy contiguous space 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.

section program counter: See SPC.

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 will be compiled or assembled to form an object file.

SPC (section program counter): An element of the assembler that keeps track of the current location within a section; each section has its own SPC.

static: A kind of 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 a symbol should be accessed.

string table: A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead, they are stored in the string table.) The name portion of a symbol's entry points to the location of the string in the string table.

structure: A collection of one or more variables grouped together under a single name.

symbol: A string of alphanumeric characters that represents an address or a value.

symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool such as a simulator or an emulator.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

T

tag: An optional “type” header name that can be assigned to a structure, union, or enumeration.

target memory: Physical memory in a TMS320C2x-, TMS320C2xx-, or TMS320C5x-based system into which executable object code is loaded.

.text: One of the default COFF sections; an initialized section that contains executable code. You can use the .text directive to assemble code into the .text section.

U

unconfigured memory: Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section: A COFF section that reserves space in the memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

union: A variable that may hold (at different times) objects of different types and sizes.

unsigned: A kind of value that is treated as a positive number, regardless of its actual sign.

V

variable: A symbol representing a quantity that may assume any of a set of values.

W

well-defined expression: An expression that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

word: A 16-bit addressable location in target memory.

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

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z